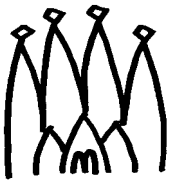


LHCb

GAUDI

LHCb Data Processing Applications Framework

User Guide

Version: 5
Issue: 1
Edition: 0
Status:
ID: [Document ID]
Date: 26 July 2000

European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	GAUDI User Guide		
	Version:	5		
	Issue:	1		
	Edition:	0		
	ID:	[Document ID]		
	Status:			
	Date:	26 July 2000		
	Keywords:			
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	5.5
	Layout Template:	Software Documentation Layout Templates	**Version:**	V1 - 15 January 1999
Authorship	**Coordinator:**	M.Cattaneo (from October 1999) P. Maley (until September 1999)		

Document Status Sheet

Table 2 Document Status Sheet

Title:	GAUDI User Guide		
ID:	[Document ID]		
Version	**Issue**	**Date**	**Reason for change**
1	0	7/Feb/99	First official (non-draft) version
1	1	10/Feb/99	Minor corrections to installation chapter
2	0	18/Jun/99	Release of version 2 of the GAUDI framework
3	0	24/Nov/99	Release of version 3 of the GAUDI framework
4	1	14/Apr/00	Release of version 4 of the GAUDI framework
5	0	24/Jul/00	Release of version 5 of the GAUDI framework
5	1	26/Jul/00	Corrections to chapters 4 and 5

Table of Contents

Chapter 1
Introduction

1.1 Purpose of the document

This document is intended as a combination of user guide and tutorial for the use of the Gaudi application framework software. It is complemented principally by two other "documents": the Architecture Design Document (ADD) [1], and the online auto-generated reference documentation [2]. A third document [3] lists the User Requirements and Scenarios that were used as input and validation of the architecture design. All these documents and other information about Gaudi, including an online version of this user guide, are available from the Gaudi home page: *http://lhcb.cern.ch/computing/Components/html/GaudiMain.html*

The ADD contains a discussion of the architecture of the framework, the major design choices taken in arriving at this architecture and some of the reasons for these choices. It should be of interest to anyone who wishes to write anything other than private analysis code.

As discussed in the ADD the application framework should be usable for implementing the full range of offline computing tasks: the generation of events, simulation of the detector, event reconstruction, testbeam data analysis, detector alignment, visualisation, etc. etc..

In this document we present the main components of the framework which are available in the current release of the software. It is intended to increment the functionality of the software at each release, so this document will also develop as the framework increases in functionality. Having said that, physicist users and developers actually see only a small fraction of the framework code in the form of a number of key interfaces. These interfaces should change very little if at all and the user of the framework cares very little about what goes on in the background.

The document is arranged as follows: Chapter 2 is a short resume of the framework architecture, presented from an "Algorithm-centric" point of view, and re-iterating only a part of what is presented in the ADD.

Chapter 3 contains a summary of the functionality which is present in the current release, and details of how to obtain and install the software.

Chapter 4 discusses in some detail an example which comes with the framework library. It covers the main program, some of the basic aspects of implementing algorithms, the specification of job options and takes a look at how the code is actually executed. The subject of coding algorithms is treated in more depth in chapter 5.

Chapter 6 discusses the use of the framework data stores. Chapters 7, 8, 9, 10 discuss the different types of data accessible via these stores: event data, detector description data (material and geometry), histogram data and n-tuples.

Chapter 11 deals with services available in the framework: job options, messages, particle properties, random numbers and also has a section on developing new services. Framework tools are discussed in Chapter 12.

A rather more technical subject, that of the use and implementation of converter classes is discussed in Chapter 13.

The use of certain SicB facilities from within Gaudi, and the use of FORTRAN code is discussed in Chapter 14. Chapter 15 gives pointers to the documentation for class libraries which we are recommending to be used within Gaudi.

Chapter 16 describes a prototype implementation of visualisation facilities inside Gaudi. Finally, Chapter 17 is a small guide to designing classes that are to be used in conjunction with the application framework.

Appendix A contains a list of references. Appendix B lists the options which may be specified for the standard components available in the current release. Appendix C gives the details of the syntax and possible error messages of the job options compiler.

1.2 Conventions

1.2.1 Units

We have decided to adopt the same system of units as CLHEP, as used also by GEANT4. This system is fully documented in the CLHEP web pages, at the URL: *http://wwwinfo.cern.ch/asd/lhc++/clhep/manual/UserGuide/Units/units.html*.

Here we reproduce the list of basic units:

- *millimetre* for length
- *nanosecond* for time
- *MeV* for energy
- *positron charge* for electric charge
- *Kelvin* for temperature
- *mole* for amount of substance
- *radian* for plane angles

- *steradian* for solid angles

Note that this differs from the convention used in SICB, where the basic units of length, time and energy are, respectively, centimetre, GeV, second.

1.2.2 Coding Conventions

The Gaudi software follows (or should follow!) the LHCb C++ coding conventions described in reference [4]. One consequence is that the specification of the C++ classes is done in two parts: the header or ".h" file and the implementation or ".cpp" file.

1.2.3 Naming Conventions

Histograms In order to avoid clashes in histogram identifiers, we suggest that histograms are placed in named subdirectories of the transient histogram store. The top level subdirectory should be the name of a sub-detector group (e.g. `/stat/VELO`). Below this, users are free to define their own structure. One possibility is to group all histograms produced by a given algorithm into a directory named after the algorithm.

1.2.4 Conventions of this document

Angle brackets are used in two contexts. To avoid confusion we outline the difference with an example.

The definition of a templated class uses angle brackets. These are required by C++ syntax, so in the instantiation of a templated class the angle brackets are retained:

```
AlgFactory<UserDefinedAlgorithm> s_factory;
```

This is to be contrasted with the use of angle brackets to denote "replacement" such as in the specification of the string:

```
"<concreteAlgorithmType>/<algorithmName>"
```

which implies that the string should look like:

```
"EmptyAlgorithm/Empty"
```

Hopefully what is intended will be clear from the context.

1.3 Reporting problems

Users of the Gaudi software are encouraged to report problems and requests for new features via the LHCb problem reporting system. This system is integrated in the CERN Problem Report Management System (CPRMS) provided by IT division, based on the Action Request System software from Remedy Corporation.

To report a new problem, go to the LHCb CPRMS home page *http://hep-service-prms.web.cern.ch/hep-service-prms/lhcb.html*, click on the **Submit** button, and fill in the form. This will add the report to the system and notify the developers by E-mail. You will receive E-mail notification of any changes in the status of your report.

To view the list of current problems, and their status, click the **Query** button on the LHCb CPRMS home page.

Active developers of the Gaudi software are encouraged to use the *gaudi-developers* mailing list for discussion of Gaudi features and future developments. This list is not, primarily, intended for bug reports. In order to send mail to *gaudi-developers@listbox.cern.ch*, you must fist subscribe to the list, using the form at *http://wwwlistbox.cern.ch/mowgli/full_list?server=listbox&group=&list=gaudi-developers*.

The archive of the mailing list is publically accessible on the Web, at *http://home.cern.ch/~majordom/news/gaudi-developers/index.html*.

1.4 Editor’s note

This document is a snapshot of the Gaudi software at the time of the release of version 5. We have made every effort to ensure that the information it contains is correct, but in the event of any discrepancies between this document and information published on the Web, the latter should be regarded as correct, since it is maintained between releases and, in the case of code documentation, it is automatically generated from the code.

We encourage our readers to provide feedback about the structure, contents and correctness of this document and of other Gaudi documentation. Please send your comments to the editor, *Marco.Cattaneo@cern.ch*

Chapter 2
The framework architecture

2.1 Overview

In this chapter we would like to briefly re-visit some of those issues addressed in the Architecture Design Document which are of particular interest to physicists wishing to use the framework. We also try to define a few of the words that are thrown around in the rest of the document, and end with some practical guidelines on software packaging.

A (more) complete view of the architecture, along with a discussion of the main design choices and the reasons for these choices may be found in reference [1].

2.2 Why architecture?

The basic “requirement” of the physicists in the collaboration is a set of programs for doing event simulation, reconstruction, visualisation, etc. and a set of tools which facilitate the writing of analysis programs. Additionally a physicist wants something that is easy to use and (though he or she may claim otherwise) is extremely flexible. The purpose of the Gaudi application framework is to provide software which fulfils these requirements, but which additionally addresses a larger set of requirements, including the use of some of the software online.

If the software is to be easy to use it must require a limited amount of learning on the part of the user. In particular, once learned there should be no need to re-learn just because technology has moved on. (You do not need to re-take your licence every time you buy a new car.) Thus one of the principal design goals was to insulate users (physicist developers and physicist analysists) from irrelevant details such as what software libraries we use for data I/O, or for graphics. We have done this by developing an architecture. An architecture consists of the specification of a number of components and their interactions with each other. A component is a “block” of software which has a well specified interface and functionality.

An interface is a collection of methods along with a statement of what each method actually does, i.e. its functionality.

We may summarise the main benefits we gain from this approach:

Flexibility This approach gives flexibility because components may be plugged together in different ways to perform different tasks.

Simplicity Software for using, for example, an object data base is in general fairly complex and time consuming to learn. Most of the detail is of little interest to someone who just wants to read data or store results. A “data access” component would have an interface which provided to the user only the required functionality. Additionally the interface would be the same independently of the underlying storage technology.

Robustness As stated above a component can hide the underlying technology. As well as offering simplicity, this has the additional advantage that the underlying technology may be changed without the user even needing to know.

It is intended that almost all software written by physicists in the collaboration whether for event generation, reconstruction or analysis will be in the form of specialisations of a few specific components. Here, specialisation means taking a standard component and adding to its functionality while keeping the interface the same. Within the application framework this is done by deriving new classes from one of the base classes:

- DataObject
- Algorithm
- Converter

In the rest of this chapter we will briefly consider the first two of these components and in particular the subject of the “separation” of data and algorithm. They will be covered in more depth in chapters 5 and 6. The third base class, Converter, exists more for technical necessity than anything else and will be discussed in chapter 13. Following this we give a brief outline of the main components that a physicist developer will come into contact with.

2.3 Data versus code

Broadly speaking, tasks such as physics analysis and event reconstruction consist of the manipulation of mathematical or physical quantities: points, vectors, matrices, hits, momenta, etc., by algorithms which are generally specified in terms of equations and natural language. The mapping of this type of task into a programming language such as FORTRAN is very natural, since there is a very clear distinction between “data” and “code”. Data consists of variables such as:

```
integer n
real p(3)
```

and code which may consist of a simple statement or a set of statements collected together into a function or procedure:

```
real function innerProduct(p1, p2)
```


```
real p1(3),p2(3)
innerProduct = p1(1)*p2(1) + p1(2)*p2(2) + p1(3)*p2(3)
end
```

Thus the physical and mathematical quantities map to data and the algorithms map to a collection of functions.

A priori, we see no reason why moving to a language which supports the idea of objects, such as C++, should change the way we think of doing physics analysis. Thus the idea of having essentially mathematical objects such as vectors, points etc. and these being distinct from the more complex beasts which manipulate them, e.g. fitting algorithms etc. is still valid. This is the reason why the Gaudi application framework makes a clear distinction between “data” objects and “algorithm” objects.

Anything which has as its origin a concept such as hit, point, vector, trajectory, i.e. a clear “quantity-like” entity should be implemented by deriving a class from the `DataObject` base class.

On the other hand anything which is essentially a “procedure”, i.e. a set of rules for performing transformations on more data-like objects, or for creating new data-like objects should be designed as a class derived from the `Algorithm` base class.

Further more you should not have objects derived from `DataObject` performing long complex algorithmic procedures. The intention is that these objects are “small”.

Tracks which fit themselves are of course possible: you could have a constructor which took a list of hits as a parameter; but they are silly. Every track object would now have to contain all of the parameters used to perform the track fit, making it far from a simple object.
Track-fitting is an algorithmic procedure; a track is probably best represented by a point and a vector, or perhaps a set of points and vectors. They are different.

2.4 Main components

The principle functionality of an algorithm is to take input data, manipulate it and produce new output data. Figure 1 shows how a concrete algorithm object interacts with the rest of the application framework to achieve this.

The figure shows the four main services that algorithm objects use:

- The event data store
- The detector data store
- The histogram service
- The message service

The particle property service is an example of additional services that can be requested by an algorithm, by asking the Application Manager to locate them via the `ISvcLocator` interface. The job options service (see Chapter 11) is used by the `Algorithm` base class, but is not usually explicitly seen by a concrete algorithm.

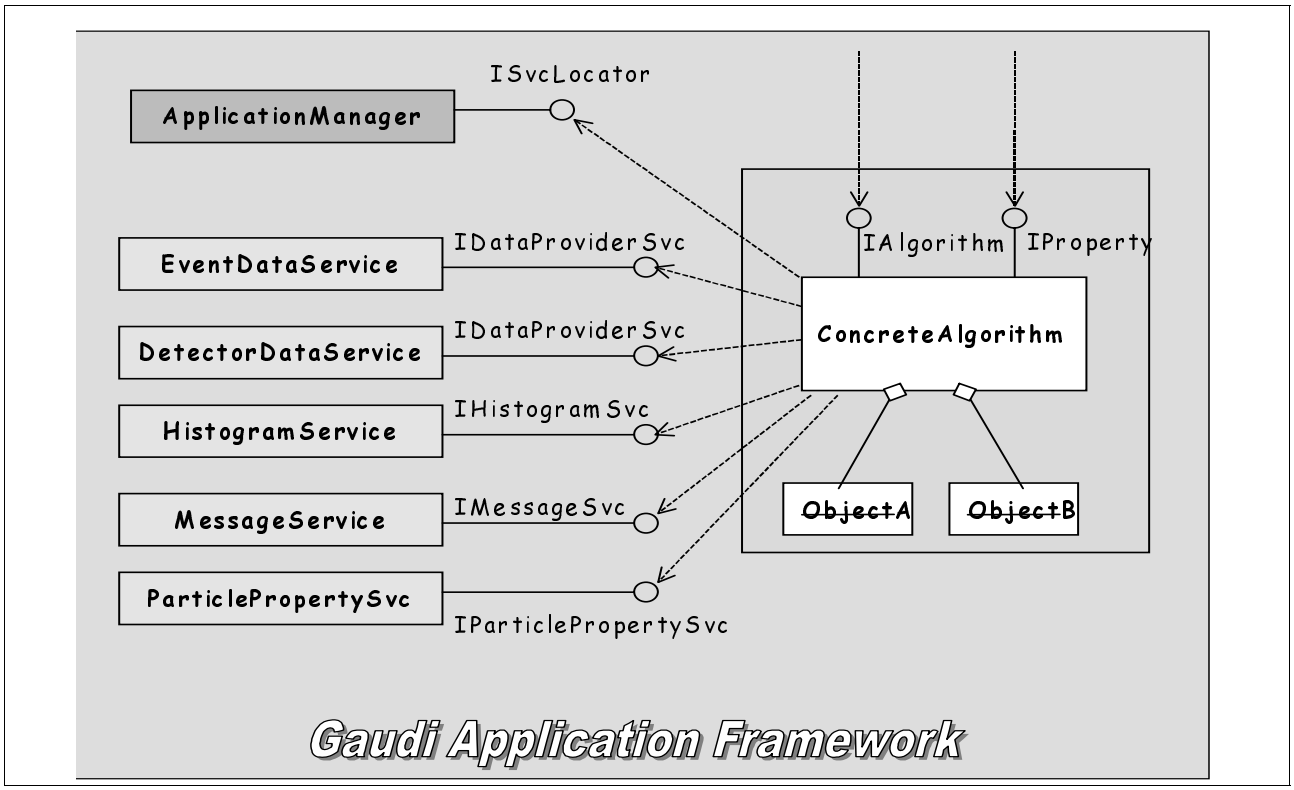


Figure 1 The main components of the framework as seen by an algorithm object.

Each of these services is provided by a component and the use of these components is via an interface. The interface used by algorithm objects is shown in the figure, e.g. for both the event data and detector data stores it is the `IDataProviderSvc` interface. In general a component implements more than one interface. For example the event data store implements another interface: `IDataManager` which is used by the application manager to clear the store before a new event is read in.

An algorithm's access to data, whether the data is coming from or going to a persistent store or whether it is coming from or going to another algorithm is always via one of the data store components. The `IDataProviderSvc` interface allows algorithms to access data in the store and to add new data to the store. It is discussed further in chapter 6 where we consider the data store components in more detail.

The histogram service is another type of data store intended for the storage of histograms and other "statistical" objects, i.e. data objects with a lifetime of longer than a single event. Access is via the `IHistogramSvc` which is an extension to the `IDataProviderSvc` interface, and is discussed in chapter 9. The n-tuple service is similar, with access via the `INtupleSvc` extension to the `IDataProviderSvc` interface, as discussed in Chapter 10.

In general an algorithm will be configurable: It will require certain parameters, such as cut-offs, upper limits on the number of iterations, convergence criteria, etc., to be initialised before the algorithm may be executed. These parameters may be specified at run time via the job options mechanism. This is done by the job options service. Though it is not explicitly shown in the figure this component makes use of the `IProperty` interface which is implemented by the Algorithm base class.

During its execution an algorithm may wish to make reports on its progress or on errors that occur. All communication with the outside world should go through the message service component via the `IMessageSvc` interface. Use of this interface is discussed in Chapter 11.

As mentioned above, by virtue of its derivation from the Algorithm base class, any concrete algorithm class implements the `IAlgorithm` and `IProperty` interfaces. `IProperty` is usually used only by the job options service.

Top level algorithms, i.e. algorithm objects created by the application manager are controlled via the `IAlgorithm` interface. This consists essentially of the methods: `initialize()`, `execute()`, and `finalize()`.

The figure also shows that a concrete algorithm may make use of additional objects internally to aid it in its function. These private objects do not need to inherit from any particular base class so long as they are only used internally. These objects are under the complete control of the algorithm object itself and so care is required to avoid memory leaks etc.

We have used the terms “interface” and “implements” quite freely above. Let us be more explicit about what we mean. We use the term interface to describe a pure virtual C++ class, i.e. a class with no data members, and no implementation of the methods that it declares. For example:

```
class PureAbstractClass {
  virtual method1() = 0;
  virtual method2() = 0;
}
```

is a pure abstract class or interface. We say that a class implements such an interface if it is derived from it, for example:

```
class ConcreteComponent: public PureAbstractClass {
  method1() { }
  method2() { }
}
```

A component which implements more than one interface does so via multiple inheritance, however, since the interfaces are pure abstract classes the usual problems associated with multiple inheritance do not occur.

Within the framework every component, i.e. services and algorithms, has two qualities:

- A concrete component class, e.g. TrackFinderAlgorithm or MessageSvc.
- Its name, e.g. “KalmanFitAlgorithm” or “stdMessageService”.

In addition, as discussed above, a component may implement several interfaces. These interfaces are identified by a unique number which is available via a global constant of the form: `IID_InterfaceType`, such as for example `IID_IDataProviderSvc`. Using these it is possible to enquire what interfaces a particular component implements.

2.5 Package structure

For large software systems, such as ours, it is clearly important to decompose the system into hierarchies of smaller and more manageable entities. This decomposition can have important consequences for implementation related issues, such as compile-time, link dependencies, configuration management, etc. For that we need to introduce the concept of *package* as the grouping of related components together into a cohesive physical entity. A package is also the minimal unit of software release.

We have decomposed the LHCb data processing software into the packages shown in Figure 2. At the lower level we find *Gaudi* which is the framework itself and only depends on some basic standard packages (STL,...). In the second level there are the packages for the specific `LHCb event` and `detector` data models. These packages depend on the framework and CLHEP. In the same level we have a specific implementation of the Histogram persistency service based on HBOOK (`HbookCnv`). When other implementations will exist, they will be added as packages. At the next level we will have packages consisting of implementations of event and detector data persistency services and converters. Currently in this release we have one based on SicB and ZEBRA (`SicbCnv`) and a generic one (`DbCnv`) which currently understands ROOT I/O and is being tested with ODBC compliant databases and with Objectivity/DB. The algorithms that will constitute the core of the data processing applications (trigger, reconstruction, analysis, etc.) will form a number a independent packages; some commony used algorithms are available in *GaudiAlg*. Finally, at the top level we find the applications.

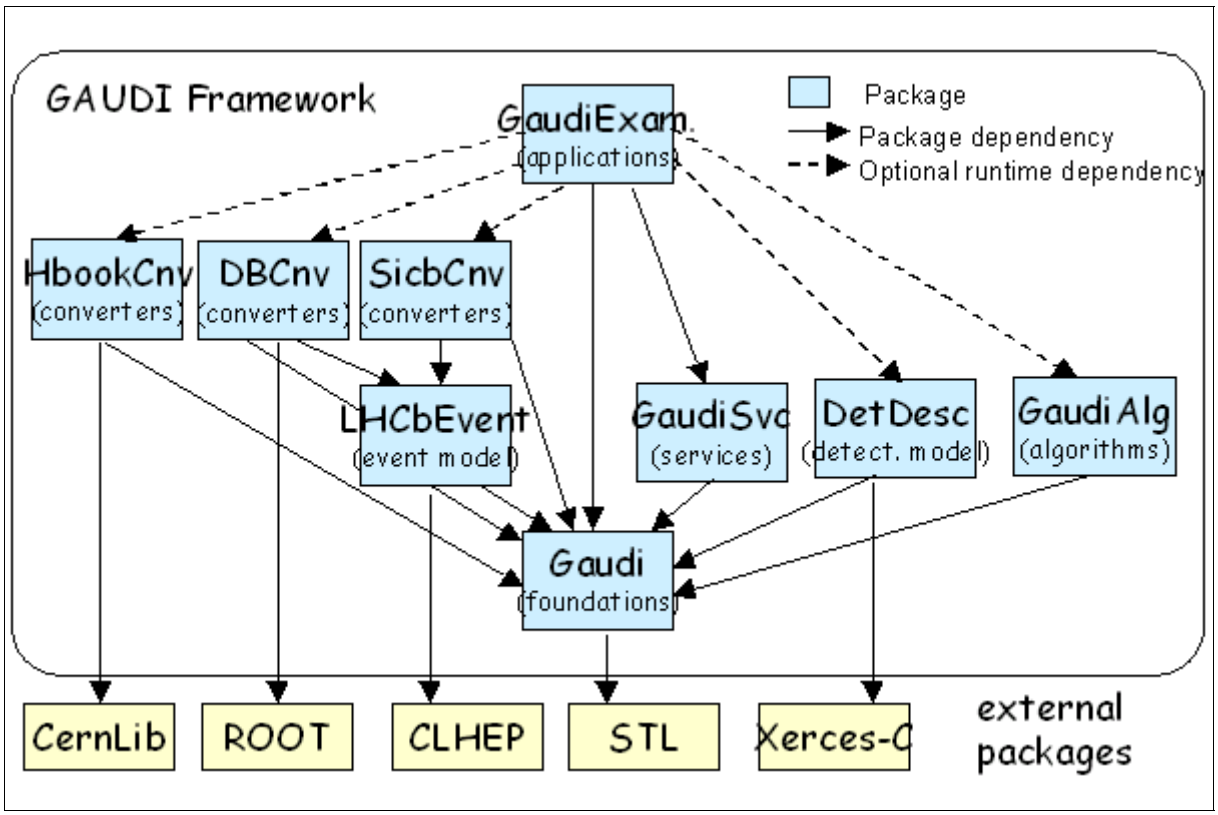


Figure 2 Current package structure of the LHCb software

2.5.1 Package Layout

Figure 3 shows the recommended layout for Gaudi packages (and for other LHCb software packages, such as sub-detector software).

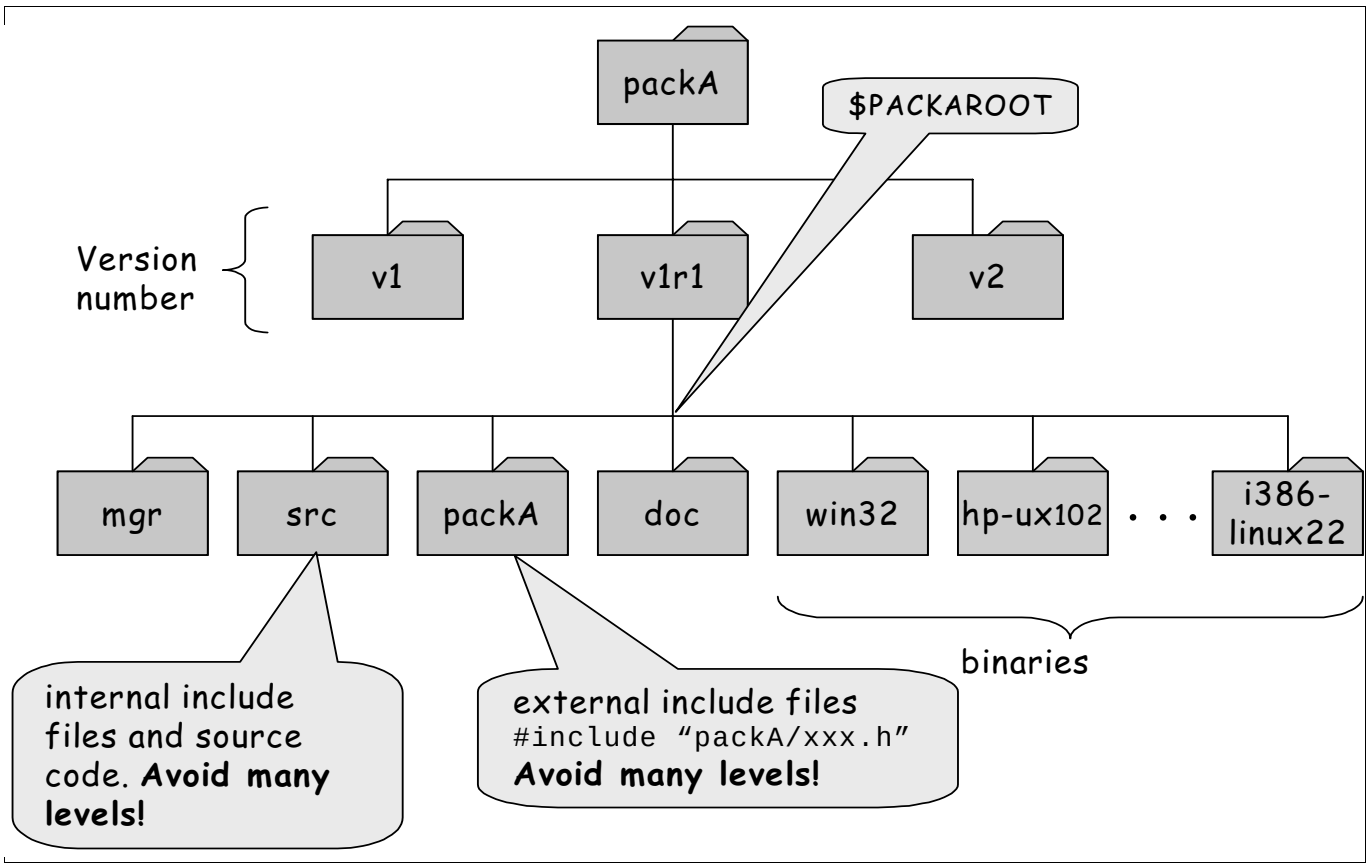


Figure 3 Recommended layout of LHCb software packages

2.5.2 Packaging Guidelines for sub-detectors

Packaging is an important architectural issue for the Gaudi framework, but also for the sub-detector software packages based on Gaudi. Typically, sub-detector packages consist of:

- Specific event model
- Specific detector description
- Sets of algorithms (digitisation, reconstruction, etc.)

The packaging should be such as to minimise the dependencies between packages, and must absolutely avoid cyclic dependencies. The granularity should not be too small or too big. Care should be taken to identify the external interfaces of packages: if the same interfaces are shared by many packages, they should be promoted to a more basic package that the others would then depend on. Finally, it is a good idea to discuss your packaging with the librarian and / or architect.

Note that sub-detectors are expected to organise their own project releases, for internal group consumption and integration tests, using the $LHCBDEV development area (described in Section 3.6.1). Once the code is ready for public, collaboration wide use, it should be released (by the librarian) in the public release area ($LHCBSOFT).

Chapter 3
Release notes and software installation

3.1 Release History

Version	Date	Package List
v5	24/07/2000	Gaudi[v7], GaudiSvc[v3], GaudiAlg[v1], LHCbEvent[v7r1], SicbCnv[v8], DbCnv[v2], HbookCnv[v6r1], DestDesc[v4], GaudiExamples[v8]
v4	14/04/2000	Gaudi[v6], GaudiSvc[v2], LHCbEvent[v7], SicbCnv[v7], DbCnv[v1], HbookCnv[v6], DestDesc[v3], GaudiExamples[v7]
v3	19/11/1999	Gaudi[v5], LHCbEvent[v6], SicbCnv[v6], RootCnv[v3], HbookCnv[v5], DestDesc[v2], GaudiLab[v1], GaudiExamples[v6]
v2	18/6/1999	Gaudi[v4], LHCbEvent[v5], SicbCnv[v5], RootCnv[v2], HbookCnv[v4], DestDesc[v1], GaudiExamples[v5]
v1	5/2/1999	Gaudi[v1], LHCbEvent[v1], SicbCnv[v1], HbookCnv[v1], GaudiExamples[v1]

3.2 Current Functionality

We use an incremental and iterative approach for producing the Gaudi software. We started in the first release with very basic functionality and we plan to expand its capabilities release by release. The basic functionality offered in the first release allowed you to read the already generated SicB Monte Carlo data sets (Zebra files). The data in these files is used to build C++ objects which can be used as input for the new software.

Release v3 incorporated most of the essential features which are needed for developing data processing applications (reconstruction, analysis, etc.). This functionality was enhanced in Release v4 and has been further enhanced in the current release. The functionality list that follows is organized by categories.

Interfaces A set of interfaces that facilitates the interaction between the different components of the framework. Mainly these are interfaces to services.

Basic framework services This set of services offer the minimal functionality needed for constructing applications. They are described in detail in Chapter 11.
The *message service* is used to send and format messages generated in the code, with an associated severity that is used for filtering and dispatching them.
The *job options service* allows the configuration of the application by end users assigning values to properties defined within the code; properties can be basic types (`float, bool, int, string`), or extended with bounds checking, hierarchical lists, and immediate callback from string "commands".
The *Random Numbers service* makes available several random number distributions via a standard interface, and ensures that applications use a unique random number engine in a reproducible fashion.
The *Chrono service* offers the functionality for measuring elapsed time and job execution statistics.
The *Tools service*, which provides management of Tools, is discussed in Chapter 12. Tools are lightweight objects which can be requested and used many times by other components to perform well defined tasks.
Data services provide the access to the transient data objects (event, detector and statistical data). The data services are described in chapters 6 to 10.

Event data model The event model has been extended to Velo, L0, L1 and updated for Calo. The current status is presented in Chapter 7.

Event data persistent storage The current version supports event data files (or tapes) as input produced by SicB in Zebra format (RZ). The Zebra banks are *converted* to C++ objects by specialized fragments of code (SicB converters). This conversion is available in the framework for a number of SicB banks. Adding more converters is possible and will be done on request. The event data produced by the application can be stored in ROOT files (Root I/O) and retrieved later. This also requires to write specialized converters. The *DBCnv* package is included to help the user to make objects persistent. Refer to Chapter 13 for more details.

Histograms & N-tuples The framework provides facilities for creating histograms (1 and 2 dimensional) and n-tuples (row and column wise) from user algorithms. The histogram interface is the AIDA[1] common interface. Saving histograms and n-tuples is currently implemented using the HBOOK format. Other persistent representations will be made available if requested. The interface to histograms and N-tuples from the user code should not be affected if the persistency representation is changed later. Details of the histogram and n-tuple facilities can be found in Chapter 9 and 10 respectively.

Detector description and geometry The framework provides facilities for describing the logical structure of the detector in terms of a hierarchy of detector elements and also the basic geometry description in terms of volumes, solids and materials. Facilities for customizing the generic description to many specific detector needs are also provided. This should allow to develop detector specific code which can provide geometry answers to questions from the physics algorithms (simulation, reconstruction and analysis). The persistent representation of the detector description is based on text files in XML format. a detailed description can be found in Chapter 8. A transport service is provided to estimate the amount of material between 2 arbitrary points in the detector setup.

1. Abstract Interfaces for Data Analysis (*http://wwwinfo.cern.ch/asd/lhc++/AIDA/index.html*)

Analysis services A number of facilities and services are included in the current release to facilitate writing physics analysis code. The *GaudiAlg* package is a collection of general purpose algorithms, including a sequencer which uses the new filtering capability of algorithms to manage the execution of algorithm sequences in a filtering application, as discussed in section 5.5. The *Particle Properties service* (section 11.5) provides the properties of all the elementary particles. Access to the CLHEP and NAG C libraries (Chapter 15) provides numerical utilities for writing physics algorithms.

Visualization services The framework provides very basic facilities (at the prototype level) for visualization of event and detector data. These services are currently based on the packages that constitutes the Open Scientist suite (OpenGL, OpenInventor(soFree), Lab,...) The idea is that any data object in the event or detector store can be represented graphically by providing a specific graphical converter. A prototype implementation of an interactive graphical user interface service is included in the release.

SicB services A number of services are included in the current release to facilitate the access to SicB data (detector description, magnetic field, etc.) and to facilitate re-using existing Fortran code with the framework. These services can be extended to accommodate new concrete requests when integrating big parts of the current legacy code (SicB). In the latest release there full support for event pile-up. It is possible to read events from two input data streams and merge them before any processing is done. Refer to Chapter 14 for more details.

Dynamic loading of libraries The framework will be used to implement different data processing applications for different environments. It is important that the services and their concrete implementations are extendable and configurable dynamically at run time and not statically. The latter would imply linking with all the available libraries producing huge executables. And in certain applications and environments some of the libraries will surely never be used. This release provides support for dynamic libraries for the NT and Linux platforms.

3.3 Availability

The application framework is supported on the following platforms:

- Windows NT using the developer studio C++ environment
- RedHat Linux 6.1 (certified CERN Linux distribution with SUE and AFS).

Support for IBM AIX was dropped as of release 3. Support for HP-UX 10.20 was dropped as of release 4.

The code, documentation and installation instructions are available from the LHCb web site at: *http://lhcb.cern.ch/computing/Components/html/GaudiMain.html*

3.4 Using the framework on NT with Developer Studio

The Framework sources are available via AFS or can be downloaded from the web. The libraries for NT are available in the LHCb AFS tree, in the *Win32Debug* subdirectory of each package.

Instructions for installing the LHCb software environment on NT and for customising MS Visual Studio can be found at: *http://lhcb.cern.ch/computing/Support/html/DevStudio/default.htm*

3.5 Using the framework in Unix

The framework libraries have been built for various Unix platforms and are available via AFS. These libraries have been built using the Configuration Management Tool (CMT). Therefore, using the CMT tool is the recommended way to modify existing packages or re-build the examples included in the release. Complete and up to date information about using CMT is available at the URL: *http://lhcb.cern.ch/computing/Support/html/cmt.htm*. Here we give only simple instructions on how to get started with the Gaudi software under CMT on Unix.

Getting a copy of a package: Suppose you want to build the latest released version of the `GaudiExamples` package:

```
[1]  cd mycmt
[2]  getpack GaudiExamples
       following versions are available - v6 v5r1 v5 v4 v3
       Enter version number [<CR>=v6]->
```

Building and running an example: Now that you have the code, suppose you want to modify the `Histograms` example, then build it and run it:

```
[3] cd GaudiExamples/v6/Histograms
[4] emacs HistoAlgorithm.cpp
----- Make your modification, then
[5] cd ../mgr
[6] emacs requirements
----- Uncomment Histograms and comment all the others
[7] gmake
[8] cd ../Histograms
[9] emacs jobOptions.txt
----- Make any modification if needed
[10] ../$CMTCONFIG/Histograms.exe
```


Modifying a library and rerunning the example: Suppose now you want to modify one of the Gaudi libraries, build it, then relink the `Histograms` example and run it:

```
[11] cd $HOME/mycmt
[12] getpack HbookCnv v5
[13] cd HbookCnv/v5/HbookCnv
[14] emacs ....Make your modification...
[15] cd ../mgr
[16] gmake
[17] cd $HOME/mycmt/GaudiExamples/v6/mgr
[18] cmt show uses
----- Make sure that you are using the modified version (v5) of
----- HbookCnv. If not, edit the requirements file to use this version
[19] gmake
[20] cd ../Histograms
[21] ../$CMTCONFIG/Histograms.exe
```

3.6 Working with development releases

This User Guide corresponds to release 5 of the GAUDI software. Subsequent to this release, new features may be added to the CVS repository which will not become generally available (and documented) until the next release. Should you wish to use some of these new features before then, you can pick them up from the *development release area.*

3.6.1 The development release area

Development releases of packages are made periodically into the AFS directory tree below `/afs/cern.ch/lhcb/software/DEV` (pointed to by the LHCBDEV environment variable on Unix), by checking out and building the CVS head revision of modified packages. The directory structure of LHCBDEV is analogous to LHCBSOFT - there is a subdirectory per package, below which there are subdirectories for each development release version of that package. These subdirectories have a name which reflects the date of the build, e.g. *h000626* was built on 26th June 2000. The most recent successful build is pointed to by a logical link whose version number is the current release version number incremented by one. For example, if the current released version of Gaudi is in `$LHCBSOFT/Gaudi/v7`, the most recent successfully built development version is in `$LHCBDEV/Gaudi/v8`.

3.6.2 Using the development version of packages

The directory structure described above was designed to make it easy to work with the development version of packages, without needing to make a private copy of the head revision of a package.

On Unix it is sufficient to modify the CMTPATH environment variable to include the development area:

```
setenv CMTPATH $HOME/mycmt:$LHCBDEV
```

Packages will then be searched first in your private CMT area, then on LHCBDEV and finally on LHCBSOFT. Note that this will always give you the latest working version of *all* packages. If you only need a specific development version of one package while keeping the production version of all other packages, it would be better to leave CMTPATH pointing only to your private area, and making a logical link to the appropriate development package. For example:

```
cd $HOME/mycmt/Gaudi
ln -fs v7 $LHCBDEV/Gaudi/h000626
```

This would allow you to use the development version of Gaudi built on 26th June 2000, while using the released version of everything else.

On NT, only the first option is available, due to the absence of logical links. You can make a search list for your CMTPATH by simply adding a second path to the *HKEY_CURRENT_USER/SOFTWARE/CMT/path* registry key.

If you only want the development version of a specific package, the only option is currently to copy the appropriate package development subdirectory into your private CMT area.

Chapter 4
Getting started

4.1 Overview

In this chapter we walk through one of the example applications (`Histograms`) which are distributed with the framework. We look briefly at the different files and go over the steps needed to compile and execute the code. We also outline where various subjects are covered in more detail in the remainder of the document. Finally we cover briefly the other example applications which are distributed and say a few words on what each one is intended to demonstrate.

4.2 Creating a job

Traditionally, a “job” is the running of a program on a specified set of input data to produce a set of output data, usually in batch.

For the example applications supplied this is essentially a two step process. First the executable must be produced, and secondly the necessary environment variables must be set and the required job options specified, as illustrated in Figure 4

The example applications consist of a number of “source code” files which together allow you to generate an executable program. These are:

- The main program.
- Header and implementation files for each of the concrete algorithm classes.
- A CMT requirements file.
- The set of Gaudi libraries.

.

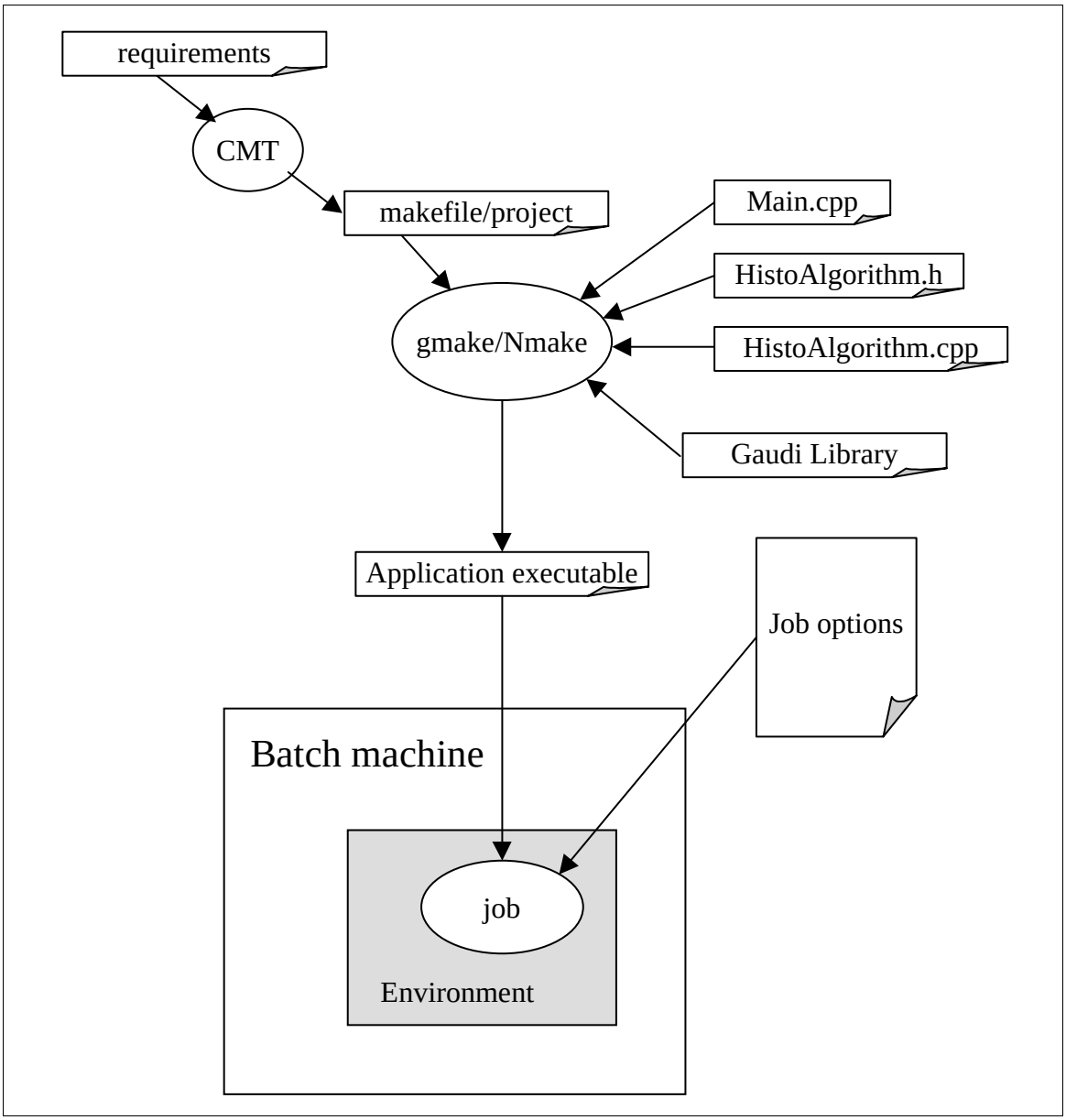


Figure 4 Creating a job from the `Histogram` example application

In order for the job to run as desired you must provide the correct configuration information for the executable. This is done via entries in the job options file.

4.3 The main program

An example main program is shown in Listing 1 on page 28. It is constructed as follows:

Include files These are needed for the creation of the application manager and Smart interface pointers.

Application Manager instantiation Line 18 instantiates an `ApplicationMgr` object. The application manager is essentially the job controller. It is responsible for creating and correctly initialising all of the services and algorithms required, for looping over the input data events and executing the algorithms specified in the job options file, and for terminating the job cleanly.

Retrieval of Interface pointers The code on lines 20 and 21 retrieves the pointers to the `IProperty` and `IAppMgrUI` interfaces of the application manager.

Setting the application manager's properties The only property which needs to be set explicitly in the main program is the name of the job options file which contains all of the other configuration information needed to run the job. This is done on line 25.

Program execution All of the code before line 33 is essentially for setting up the job. Once this is done, a call to `ApplicationMgr::run()` is all that is needed to start the job proper! The steps that occur within this method are discussed briefly in section 4.6

4.4 Configuring the job

The application framework makes use of a job options file (or possibly a database in future) for job configuration. The job options file of the `Histograms` example application is shown in Listing 2 on page 29.

The format of an options file is discussed fully in Chapter 11. Options may be set both for algorithms and services and the list of available options for standard components is given in Appendix B.

For the moment we look briefly at one of the options. The option `TopAlg` of the application manager is a list of algorithms that will be created and controlled directly by the application manager, the so-called top-level algorithms. The syntax is a list of the form:

```
ApplicationMgr.TopAlg = { "Type1/Name1","Type2/Name2" };
```

The line above instructs the application manager to make two top level algorithms. One of type `Type1` called "`Name1`" and one of type `Type2` called "`Name2`". In the case where the name of the algorithm is the same as the algorithm's type (i.e. class), only the class name is necessary. Thus, in the example in Listing 2 line 37, an instance of the class `"HistoAlgorithm"` will be created with name `"HistoAlgorithm"`

Listing 1 The example main program.

```
 1:  // Include files
 2:  #include "Gaudi/Kernel/SmartIF.h"
 3:  #include "Gaudi/Kernel/Bootstrap.h"
 4:  #include "Gaudi/Interfaces/IAppMgrUI.h"
 5:  #include "Gaudi/Interfaces/IProperty.h"
 6:  #include "Gaudi/JobOptionsSvc/Property.h"
 7:  //--------------------------------------------------------------------
 8:  //  Package    : Gaudi Examples
 9:  //  Description: Main Program
10:  //--------------------------------------------------------------------
11:
12:  //--- Example main program
13:  int main(int argc, char* argv[]) {
14:
15:    StatusCode status = StatusCode::SUCCESS;
16:
17:  // Create an instance of an application manager
18:    IInterface* iface = Gaudi::createApplicationMgr();
19:
20:    SmartIF<IProperty>     propMgr ( IID_IProperty, iface );
21:    SmartIF<IAppMgrUI>     appMgr  ( IID_IAppMgrUI, iface );
22:
23:    // Set properties of algorithms and services
24:    if ( propMgr == iface )    {
25:      status = propMgr->setProperty( StringProperty("JobOptionsPath",
     "jobOptions.txt") );
26:    }
27:    else  {
28:      exit(0);
29:    }
30:
31:    // Run the application manager and process events
32:    if ( appMgr )   {
33:      status = appMgr->run();
34:    }
35:    else  {
36:      return 0;
37:    }
38:
39:    // All done - exit
40:    return 0;
41:  }
42:
```


Listing 2 The job options file for the `Histograms` example application.

```
 1: //==============================================================
 2: //
 3: // Histograms job options file
 4: //
 5: //==============================================================
 6:
 7: // For Windows/NT
 8: #ifdef WIN32
 9: #include "../Common/Win32StandardJob.txt"
10: #else
11: // For Unix
12: #include "../Common/UnixStandardJob.txt"
13: #endif
14:
15: //--------------------------------------------------------------
16: // Event related parameters
17: //--------------------------------------------------------------
18: // Number of events to be processed (default is 10)
19: EventSelector.EvtMax = 100;
20: // Print event number every event
21: EventSelector.PrintFreq = 1;
22:
23: //--------------------------------------------------------------
24: // Input file
25: //--------------------------------------------------------------
26: #ifdef WIN32
27: // For Windows/NT:
28: ApplicationMgr.EvtSel = "FILE
    $AFSROOT\cern.ch\lhcb\data\mc\sicb_bpipi_v233_100ev.dst1";
29: #else
30: // For UNIX:
31: ApplicationMgr.EvtSel = "JOBID 16434";
32: #endif
33:
34: //--------------------------------------------------------------
35: // Private Application Configuration options
36: //--------------------------------------------------------------
37: ApplicationMgr.TopAlg = { "HistoAlgorithm" };
38: //--------------------------------------------------------------
39: // Set output level threshold (2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR,
    6=FATAL )
40: //--------------------------------------------------------------
41: MessageSvc.OutputLevel      = 3;
42: HistoAlgorithm.OutputLevel  = 3;
43: //--------------------------------------------------------------
44: // Histogram output file
45: //--------------------------------------------------------------
46: HistogramPersistencySvc.OutputFile  = "histo.hbook";
47: //--------------------------------------------------------------
48: // Algorithms Private Options
49: //--------------------------------------------------------------
50: // 0 = false = no histograms
51: HistoAlgorithm.HistogramFlag = true;
```


4.5 Algorithms

The subject of specialising the Algorithm base class to do something useful will be covered in detail in chapter 5. Here we will limit ourselves to looking at the class `HistoAlgorithm` in the example just to get an idea of the basics. For the full listing of this class, refer to the software distribution. Here we look only at those parts of the code which do something interesting and leave the technicalities for later.

4.5.1 The `HistoAlgorithm` header file

The `HistoAlgorithm` class definition is shown in Listing 3.

Note the following:

- The class is derived from the Algorithm base class as must be all specialised algorithm classes. This implies that the `Algorithm.h` file must be included.
- All derived algorithm classes must provide a constructor with the parameters shown in line 8. The first parameter is the name of the algorithm and is used amongst other things to locate any options that may have been specified in the job options file.
- The `HistoAlgorithm` class has three (private) data members, defined in lines 18 to 21. These are a boolean flag, and two pointers to histogram objects.
- The three methods on lines 11 to 13 must be implemnted, since they are pure virtual in the base class.

Listing 3 The header file of the class: `HistoAlgorithm`.

```
 1: #include "Gaudi/Algorithm/Algorithm.h" // Required for inheritance
 2: class IHistogram1D;                   // Forward declaration
 3: class IHistogram2D;                   // Forward declaration
 4:
 5: class HistoAlgorithm : public Algorithm {
 6: public:
 7: // Constructor of this form must be provided
 8: HistoAlgorithm(const std::string& name, ISvcLocator* pSvcLocator);
 9:
10: // Three mandatory member functions of any algorithm
11: StatusCode initialize();
12: StatusCode execute();
13: StatusCode finalize();
14:
15: private:
16: // These data members are used in the execution of this algorithm
17: // They are set in the initialisation phase by the job options service
18:   bool            m_produceHistogram;
19:   // Two histograms ( used if m_produceHistogram = 1 (true) )
20:   IHistogram1D*   m_hTrackCount;
21:   IHistogram2D*   m_hPtvsP;
22: };
```


4.5.2 The `HistoAlgorithm` implementation file

The implementation file contains the actual code for the constructor and for the methods: `initialize()`, `execute()` and `finalize()`. It also contains two lines of code for the `HistoAlgorithm` factory, which we will discuss in section 5.3.1

The constructor must call the base class constructor, passing on its two arguments. As usual, member variables should be initialised. Here we set the `m_produceHistogram` flag to false and set the histogram pointers to zero. This is also the place to declare any member variables that you wish to be set by the job options service. This is done by making a call to the `declareProperty()` method.

```
1:  HistoAlgorithm::HistoAlgorithm(const std::string& name,
2:                                 ISvcLocator* pSvcLocator) :
3:                               Algorithm(name, pSvcLocator),
4:                               m_hTrackCount(0), m_hPtvsP(0) {
5:  // Declare the algorithm's properties
6:  declareProperty( "HistogramFlag", m_produceHistogram = false ); }
```

Initialisation The application manager invokes the `sysInitialize()` method of the algorithm base class which, in turn, invokes the `initialize()` method of the base class, the `setProperties()` method, and finally the `initialize()` method of the concrete algorithm class. As a consequence all of an algorithm's properties will have been set before its `initialize()` method is invoked, and all of the standard services such as the message service are available. This is discussed in more detail in chapter 5.

Looking at the code in the example we see that depending on the value of the histogram flag the `HistoAlgorithm` class may also create two histograms at initialisation and declare them to the histogram data service (see Chapter 9 for details on the use of histograms in Gaudi):

Listing 4 Example of creation of histograms

```
 1:  if( m_produceHistogram )   {
 2:    log << MSG::INFO << "Histograms will be produced" << endreq;
 3:
 4:    m_hTrackCount = histoSvc()->book( "/stat/simple1D/1", "TrackCount",
     100,0,3000);
 5:    if( 0 == m_hTrackCount ) {
 6:      log << MSG::ERROR << "Cannot register histo TrackCount"<<endreq;
 7:    }
 8:    m_PtvsP = histoSvc()->book("stat/simple2D/3","Pt versus P",
     100,0,200,100,0,5);
 9:    if( 0 == m_PtvsP ) {
10:      log << MSG::ERROR << "Cannot register histo Pt versus P"<<endreq;
11:    }
12:  }
```

execution The execute method is where most of the real action takes place. This method is called by the application manager once for every event. The `HistoAlgorithm` class accesses the event data store to retrieve a container of track objects (line 2). It then fills one of the two previously created histograms with the number of particles (line 15).

In order to use the objects within the container an iterator is defined (line 19) and the second histogram is filled with the momentum of the tracks (line 24).

```
 1: // MCParticle
 2: SmartDataPtr<MCParticleVector> particles( eventSvc(),
 3:                                           "/Event/MC/MCParticles" );
 4:
 5: if( !particles ) {
 6:   log << MSG::ERROR << "Unable to retrieve MCParticles" << endreq;
 7:   return 0;
 8: }
 9: log << MSG::INFO << "+++++ MCParticles retrieved +++++" << endreq;
10:
11: // Histograms
12: if( m_produceHistogram ) {
13:
14:   // Fill the track count histogram
15:   m_hTrackCount->fill(particles->size(), 1.);
16:
17: // Fill the Pt versus P scatterplot
18:     // Iterate over all tracks in the track container
19:   MCParticleVector::iterator iterP = 0;
20:   for( iterP = particles->begin(); iterP != particles->end(); iterP++ ) {
21:       // Get the energy of the track, convert it to GeV, and histogram it
22:       double pt = (*iterP)->fourMomentum().perp();
23:       double p  = (*iterP)->fourMomentum().rho();
24:       m_hPtvsP->fill( p / GeV, pt / GeV, 1. );
25:     }
26: }
```

The details of the event data store and how data is accessed are discussed in Chapter 6.

Finalisation At finalisation, the `HistoAlgorithm` class asks the histogram service to printout the histograms in different formats.

MsgStream The `HistoAlgorithm` class makes use of the message service and the `MsgStream` utility class in order to print out some progress information, for example:

```
MsgStream          log( msgSvc(), name() );
log << MSG::INFO << "finalize" << endreq;
```

The first line creates a local `MsgStream` object, which uses the Algorithm's standard message service via the `msgSvc()` accessor, and the algorithm's name via the `name()` accessor. The use of these is discussed in more detail in Chapter 11.

4.6 Job execution

From the main program and the requirements file we can make an executable. This executable together with the file of job options form a job which may be submitted for batch or run interactively. Figure 5 shows a trace of an example program execution. The diagram is not

intended to be complete, merely to illustrate a few of the points mentioned earlier in the chapter.

1. The application manager instantiates the required services and initialises them. The message service is done first to allow the other services to use it, and the job options service is second so that the other services may be configured at run time.
2. The algorithms which have been declared to the application manager within the job options (via the `TopAlg` option) are created. We denote these algorithms “top-level” as they are the only ones controlled directly by the application manager. For illustration purposes we instantiate also an `EmptyAlgorithm` as well as the `HistoAlgorithm` of the example
3. The top-level algorithms are initialised. They may request that their properties (if they have any) are set and may make use of the message service. If any algorithm fails to initialise the job is aborted.
4. The application manager now starts to loop over events. After each event is read, it executes each of the top level algorithms in order. The order of execution of the algorithms is the order in which they appear in the `TopAlg` option. This will continue until the required number of events has been processed. If a particular algorithm returns a FAILURE status code many times, the application manager may decide that this algorithm is badly configured and jump to the finalisation stage before all events have been processed.
5. After the required data sample has been read the application manager finalises each top level algorithm.
6. Services are finalised.
7. All objects are deleted and resources freed. The program terminates.

4.7 Simple Physics Analysis Example

An example of how to do some very simple physics analysis is provided in the GaudiExamples package. The algorithms in the example use many of the Gaudi Services that someone would want to be able to utilize while doing physics analysis: histograms, ntuples, creating and retrieving private transient data, retrieving particle properties (like mass values), etc. Detailed examples on how to use the specific services are provided in the topical examples but in the SimpleAnalysis example they are combined togheter. Tools to make physics analysis in a more elegant and complex way are under development and their concrete implementation will be part of the OO Physics Analysis Program.

The SimpleAnalysisAlgorithm is an example in which pi+ pi- invariant masses are made while requiring the component particles to satisfy some simple kinematic and quality cuts. Private containers of the particles satisfying successive cuts are created and filled (charged particles, detection in the silicon, best particle ID). Invariant masses are made and corresponding histograms are filled for all combinations of the final private containers, for combinations with Pt of both pions greater than a cut value and for combinations with impact parameter of both pions greater then a cut value. The Pt and impact parameter cut values are properties of the algorithm and as such can be specified in the jobOptions, where the number is taken in Gaudi Units. CLHEP vectors' classes are used to evaluate transverse momentum

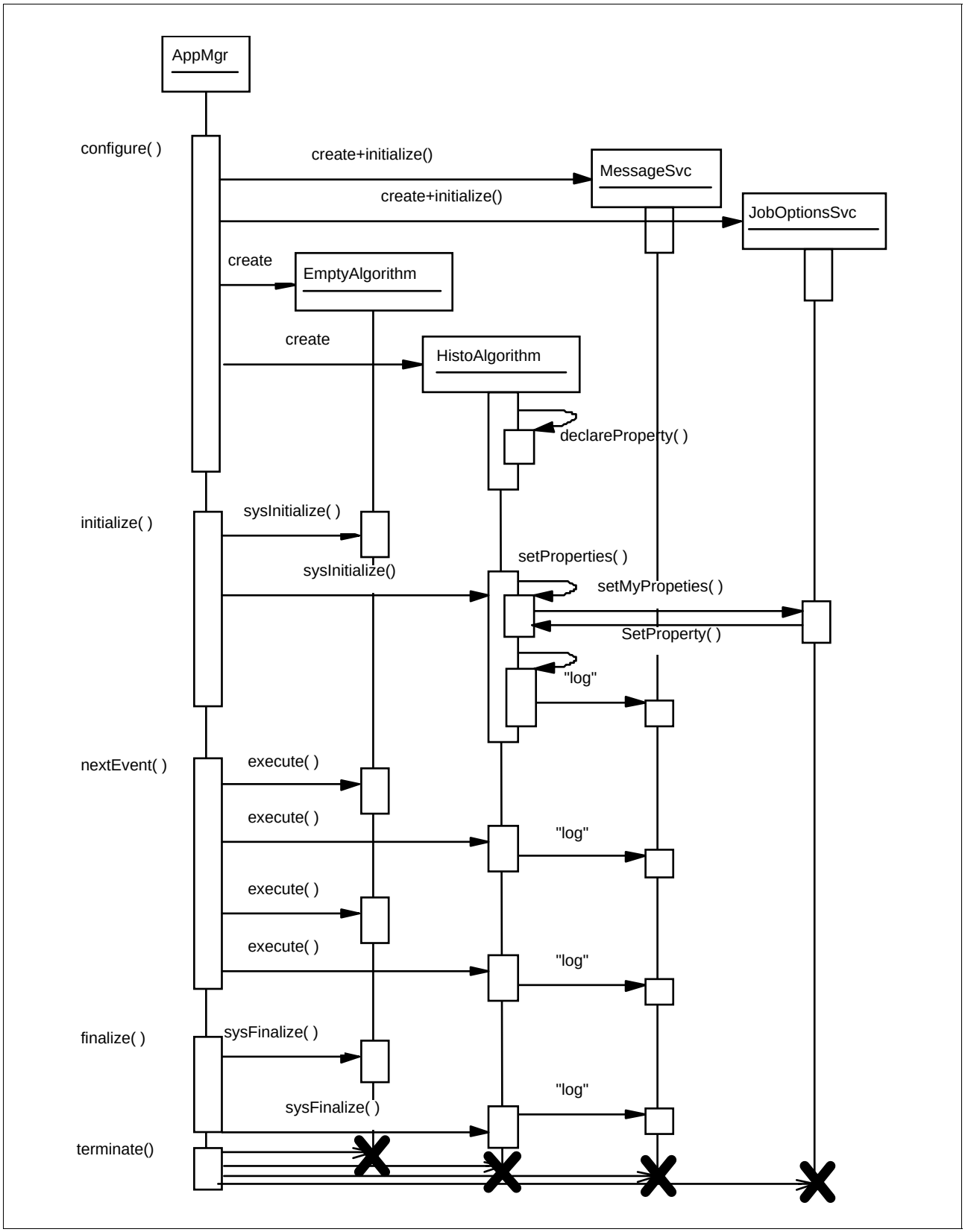


Figure 5 A sequence diagram showing a part of the execution of the example program.

and invariant masses as well as to calculate the impact parameter. When nominal mass values are required they are retrieved via the ParticlePropertySvc. Since a primary vertex is required a "dummy" algorithm RecPrimaryVertex retrieves the Monte Carlo primary vertex and uses the quantites to fill a MyVertex object (/Event/MyAxVertices), which is then retrieved by the SimpleAnalysisAlgorithm. Since the MyVertex object is created and registered in the Transient Event store by the RecPrimaryVertex algorithm, the sequencing of RecPrimaryVertex and SimpleAnalysisAlgorithm in the jobOptions file is very important. A protection is put in place so that the SimpleAnalysisAlgorithm will return a failure code if not all of the necessary input data exist in the store.

When doing physics analysis on Monte Carlo data, it is necessary to compare the reconstructed decay with the Monte Carlo truth in order to calculate efficiencies. The MCDecayFinder algorithm is an example of how to find any one step decay. The parent of the decay and the list of its direct descendants are properties of the algorithm and can be specified in the jobOptions file. If no decay is specified in the jobOptions this example will look for a B0->pi+pi- decay. The Algorithm will retrieve the particle Geant3 ID from the ParticlePropertySvc (the identifying particleID in MCParticles) and search the MCParticles to find the requested parent and that is has the correct type and number of decay products. If a decay is found kinematic variables are stored in an ntuple that can be accessed by PAW. In addition the Algorithm uses the Message service with DEBUG or INFO levels to print a summary of its behaviour for each event as well as for the job.

4.8 Other examples distributed with Gaudi

A number of examples is included in the current release of the framework. The intention is that each example exercises and shows to an end-user how to make use of some part of the functionality of the framework. The following table shows the list of available examples.

Table 3 List of available examples in current release

Example Name	Target Functionality
AlgSequencer	Illustrating the use of the sequencer algorithm provided in the GaudiAlg package
Common	Actually not a complete example: contains main program used by many examples (AlgSequencer, DDexample, DumpEvent, FieldGeom, FortranAlgorithm, Histograms, MCPrimaryVertex, ParticleProperties, SimpleAnalysis, ToolsAnalysis) and system specific Job Options include files common to all examples
DDexample	Illustrating the use of the detector description
DumpEvent	Navigation of the LHCb transient event data model
FieldGeom	Making available existing Sicb magnetic field and geometry data to Gaudi algorithms. Example of nested algorithms
FortranAlgorithm	Wrapping Fortran code in Gaudi
Histograms	Basic functionality of the framework to execute a simple algorithm, access event data and fill histograms.
MCPrimaryVertex	Retrieve data using SmartDataPtr
Ntuples	Two examples, reading and writing Ntuples
OpenScientist	Example of use of OpenScientist with Gaudi (works only on Unix at CERN)
ParticleProperties	Access the Particle Properties service to retrieve Particle Properties
RandomNumber	Example of use of the Random Number service
Rio.Example1	Two examples, reading and writing persistent data with ROOT I/O
SimpleAnalysis	A realistic example of using the framework for a physics analysis, including access to Monte Carlo data, creation of reconstructed data and filling an n-tuple
ToolsAnalysis	Example of use of framework tools in an analysis

Chapter 5
Writing algorithms

5.1 Overview

As mentioned previously the framework makes use of the inheritance mechanism for specialising the Algorithm component. In other words, a concrete algorithm class must inherit from ("be derived from" in C++ parlance, "extend" in Java) the Algorithm base class.

In this chapter we first look at the base class itself. We then discuss what is involved in creating concrete algorithms: specifically how to declare properties, what to put into the methods of the IAlgorithm interface and the use of private objects. Finally we look at how to nest algorithms.

5.2 Algorithm base class

Since a concrete algorithm object *is-an* Algorithm object it may use all of the public methods of the Algorithm base class. The base class has no protected methods or data members, and no public data members, so in fact, these are the only methods that are available. Most of these methods are in fact provided solely to make the implementation of derived algorithms easier. The base class has two main responsibilities: the initialisation of certain internal pointers and the management of the properties of derived algorithm classes.

A part of the Algorithm base class definition is shown in Listing 5. Include directives, forward declarations and private member variables have all been suppressed. It declares a constructor and destructor; the three key methods of the `IAlgorithm` interface; several accessors to services that a concrete algorithm will almost certainly require; a method to create a sub algorithm, the two methods of the `IProperty` interface; and a whole series of methods for declaring properties.

Listing 5 The definition of the Algorithm base class.

```
 1: class Algorithm : virtual public IAlgorithm,
 2:                   virtual public IProperty {
 3: public:
 4: // Constructor and destructor
 5: Algorithm( const std::string& name, ISvcLocator *svcloc );
 6: virtual ~Algorithm();
 7:
 8: StatusCode sysInitialize();
 9: //virtual StatusCode initialize();
10: StatusCode sysExecute();
11: //virtual StatusCode execute();
12: StatusCode sysFinalize();
13: //virtual StatusCode finalize();
14: const std::string&  name() const;
15:
16: virtual bool isExecuted() const;
17: virtual StatusCode setExecuted( bool state );
18: virtual StatusCode resetExecuted();
19: virtual bool isEnabled() const;
20: virtual bool filterPassed() const;
21: virtual StatusCode setFilterPassed( bool state );
22:
23: IMessageSvc*        msgSvc();
24: IDataProviderSvc*   eventSvc();
25: IConversionSvc*     eventCnvSvc();
26: IDataProviderSvc*   detSvc();
27: IConversionSvc*     detCnvSvc();
28: IHistogramSvc*      histoSvc();
29: INtupleSvc*         ntupleSvc();
30: IChronoStatSvc*     chronoSvc();
31: IRndmGenSvc*        randSvc();
32: ISvcLocator*        serviceLocator();
33: void setOutputLevel( int level );
34:
35: StatusCode createSubAlgorithm( const std::string& type,
36: const std::string& name, Algorithm*& pSubAlg );
37: std::vector<Algorithm*>* subAlgorithms() const;
38:
39: virtual StatusCode setProperty(const Property& p);
40: virtual StatusCode getProperty(Property* p) const;
41: const Property& getProperty( const std::string& name) const;
42: const std::vector<Property*>& getProperties() const;
43: StatusCode setProperties();
44:
45: StatusCode declareProperty(const std::string& name, int& reference);
46: StatusCode declareProperty(const std::string& name, float& reference);
47: StatusCode declareProperty(const std::string& name, double& reference);
48: StatusCode declareProperty(const std::string& name, bool& reference);
49: StatusCode declareProperty(const std::string& name,
50:                            std::string& reference);
51: // Vectors of properties not shown
52: private:
53: // Data members not shown
54:
55: Algorithm(const Algorithm& a);   // NO COPY ALLOWED
56: Algorithm& operator=(const Algorithm& rhs); // NO ASSIGNMENT ALLOWED
57: };
```


Constructor and Destructor The base class has a single constructor which takes two arguments: The first is the name that will identify the algorithm object being instantiated and the second is a pointer to one of the interfaces implemented by the application manager: `ISvcLocator`. This interface may be used to request special services that an algorithm may wish to use, but which are not available via the standard accessor methods (below).

The `IAlgorithm` interface Principally this consists of the three pure virtual methods that must be implemented by a derived algorithm: `initialize()`, `execute()` and `finalize()`. These are where the algorithm does its useful work and discussed in more detail in section 5.3. Other methods of the interface are the accessor `name()` which returns the algorithm's identifying name, and `sysInitialize()`, `sysFinalize()`, `sysExecute()` which are used internally by the framework. The latter three methods are not virtual and may not be overridden.

Service accessor methods Lines 23 to 32 declare accessor methods which return pointers to key service interfaces. These methods are available for use only after the Algorithm base class has been initialised, i.e. they may not be used from within a concrete algorithm constructor, but may be used from within the `initialize()` method (see 5.3.3). The services and interface types to which they point are self explanatory (see also Chapter 2). Additional services may be located using the `serviceLocator()` accessor method on line 32, as described in section 11.2 of Chapter 11. Line 33 declares a facility to modify the message output level from within the code (the message service is described in detail in section 11.4 of Chapter 11).

Creation of sub algorithms The methods on lines 35 to 37 are intended to be used by a derived class to manage sub-algorithms, as discussed in section 5.4.

Declaration and setting of properties As mentioned above, one of the responsibilities of the base class is the management of properties. The methods in lines 39 to 43 are used by the framework to set properties as defined in the job options file. The `declareProperty` methods (lines 45 to 51) are intended to be used by a derived class to declare its properties. This is discussed in more detail in section 5.3.2. and in Chapter 11.

Filtering The methods in lines 16 to 21 are used by sequencers and filters to access the state of the algorithm, as discussed in section 5.5.

5.3 Derived algorithm classes

In order for an algorithm object to do anything useful it must be specialised, i.e. it must extend (inherit from, be derived from) the `Algorithm` base class. In general it will be necessary to implement the methods of the `IAlgorithm` interface, and declare the algorithm's properties to the property management machinery of the `Algorithm` base class. Additionally there is one non-obvious technical matter to cover, namely algorithm factories.

5.3.1 Creation (and algorithm factories)

As mentioned before, a concrete algorithm class must specify a single constructor with the same parameter signature as the constructor of the base class.

In addition to this a concrete algorithm factory must be provided. This is a technical matter which permits the application manager to create new algorithm objects without having to include all of the concrete algorithm header files. From the point of view of an algorithm developer it implies adding two lines into the implementation file, of the form:

```
static const AlgFactory<ConcreteAlgorithm>  s_factory;
const IAlgFactory& ConcreteAlgorithmFactory = s_factory;
```

where "`ConcreteAlgorithm`" should be replaced by the name of the derived algorithm class (see for example lines 10 and 11 in Listing 6 below).

5.3.2 Declaring properties

In general a concrete algorithm class will have several data members which are used in the execution of the algorithm proper. These data members should of course be initialised in the constructor, but if this was the only mechanism available to set their value it would be necessary to recompile the code every time you wanted to run with different settings. In order to avoid this, the framework provides a mechanism for setting the values of member variables at run time.

The mechanism comes in two parts: the declaration of properties and the setting of their values. As an example consider the class `TriggerDecision` in Listing 6 which has a number of variables whose value we would like to set at run time.

Listing 6 Declaring member variables as properties.

```
 1: //------- In the header file -------------------------------------//
 2: class TriggerDecision : public Algorithm {
 3:
 4: private:
 5:   bool m_passAllMode;
 6:   int m_muonCandidateCut;
 7:   std::vector m_ECALEnergyCuts;
 8: }
 9: //------- In the implementation file ------------------------------//
10: static const AlgFactory<TriggerDecision>  s_factory;
11: const IAlgFactory& TriggerDecisionFactory = s_factory;
12:
13: TriggerDecision::TriggerDecision(std::string name, ISvcLocator *pSL) :
14:   Algorithm(name, pSL), m_passAllMode(false), m_muonCandidateCut(0) {
15:   m_ECALenergyCuts.push_back(0.0);
16:   m_ECALenergyCuts.push_back(0.6);
17:
18:   declareProperty("PassAllMode", m_passAllMode);
19:   declareProperty("MuonCandidateCut", m_muonCandidateCut);
20:   declareProperty("ECALEnergyCuts", m_ECALEnergyCuts);
21: }
22:
23: StatusCode TriggerDecision::inialize() {
24: }
```


The default values for the variables are set within the constructor (within an initialiser list) as per normal. To declare them as properties it suffices to call the `declareProperty()` method. This method is overloaded to take an `std::string` as the first parameter and a variety of different types for the second parameter. The first parameter is the name by which this member variable shall be referred to, and the second parameter is a reference to the member variable itself.

In the example we associate the name "`PassAllMode`" to the member variable `m_passAllMode`, and the name "`MuonCandidateCut`" to `m_muonCandidateCut`. The first is of type boolean and the second an integer. If the job options service (described in Chapter 11) finds an option in the job options file belonging to this algorithm and whose name matches one of the names associated with a member variable, then that member variable will be set to the value specified in the job options file.

5.3.3 Implementing IAlgorithm

In order to implement `IAlgorithm` you must implement its three pure virtual methods `initialize()`, `execute()` and `finalize()`. For a top level algorithm, i.e. one controlled directly by the application manager, the methods are invoked as is described in section 4.6. This dictates what it is useful to put into each of the methods.

Initialisation In a standard job the application manager will initialise all top level algorithms exactly once before reading any event data. It does this by invoking the `sysInitialize()` method of each top-level algorithm in turn. Figure 6 shows an example trace of the initialization phase. Sub-algorithms are discussed in section 5.4.

The framework takes care of setting up internal references to standard services. The algorithm properties are set before the `initialize()` method is called by calling the method `Algorithm::setProperties()`. This makes a call to one of the methods of the job options service passing a reference to itself as a parameter. The job options service then makes repeated calls to the `setProperty()` method of the algorithm which actually assigns values to the member variables.

The `initialize()` method can be used to do such things as creating histograms, or creating sub-algorithms if required (see section 5.4).

If an algorithm fails to initialise it should return `StatusCode::FAILURE`. This will cause the job to terminate.

Execution The guts of the algorithm class is in the `execute()` method. For top level algorithms this will be called once per event for each algorithm object in the order in which they were declared to the application manager. For sub-algorithms (see section 5.4) the control flow may be as you like: you may call the `execute()` method once, many times or not at all.

Just because an algorithm derives from the `Algorithm` base class does not mean that it is limited to using or overriding only the methods defined by the base class. In general, your code will be much better structured (i.e. understandable, maintainable, etc.) if you do not, for example, implement the `execute()` method as a single block of 100 lines, but instead define your own utility methods and classes to better structure the code.

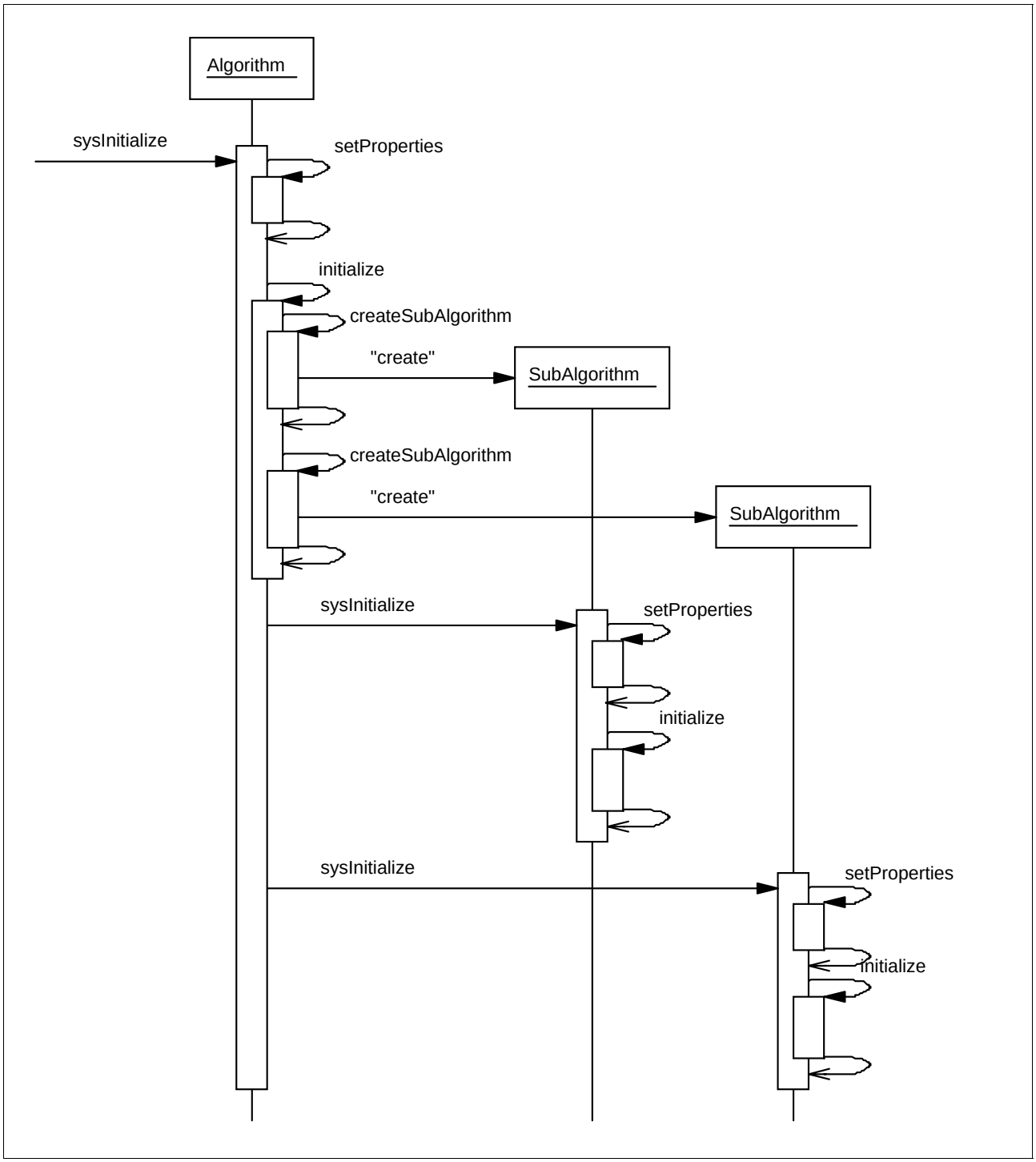


Figure 6 Algorithm initialization.

If an algorithm fails in some manner, e.g. a fit fails to converge, or its data is nonsense it should return from the `execute()` method with `StatusCode::FAILURE`. This will cause the application manager to stop processing events and end the job. This default behaviour can be modified by setting the `<myAlgorithm>.ErrorMax` job option to something greater than 1. In this case a message will be printed, but the job will continue as if there had been no error,

and just increment an error count. The job will only stop if the error count reaches the `ErrorMax` limit set in the job option.

The framework (the Algorithm base class) calls the `execute()` method within a try / catch clause. This means that any exception not handled in the execution of an Algorithm will be caught at the level of `sysExecute()` implemented in the base class. The behaviour on these exceptions is identical to that described above for errors. The measurement of elapsed CPU on each execution of an algorithm is also implemented by default using the chrono service (described in Chapter 11). This can be turned on or off with the boolean properties `ProfileInitialize, ProfileExecute, ProfileFinalize`.

Finalisation The `finalize()` method is called at the end of the job. It can be used to analyse statistics, fit histograms, or whatever you like. Similarly to initialization, the framework invokes a sysFinalize method which in turn invokes the `finalize()` method of the algorithm and of any sub-algorithms.

The following is a list of things to do when implementing an algorithm.

- Derive your algorithm from the `Algorithm` base class.
- Provide the appropriate constructor and the three methods `initialize()`, `execute()` and `finalize()`.
- Make sure you have implemented a factory by adding the magic two lines of code (see 5.3.1).

5.4 Nesting algorithms

The application manager is responsible for initialising, executing once per event, and finalising the set of top level algorithms, i.e. the set of algorithms specified in the job options file. However such a simple linear structure is very limiting. You may wish to execute some algorithms only for specific types of event, or you may wish to “loop” over an algorithm’s execute method. Within the LHCb application framework the way to have such control is via the nesting of algorithms. A nested (or sub-) algorithm is one which is created by, and thus belongs to and is controlled by, another algorithm (its parent) as opposed to the application manager. In this section we discuss a number of points which are specific to sub-algorithms.

In the first place, the parent algorithm will need a member variable of type `Algorithm*` (see the code fragment below) in which to store a pointer to the sub-algorithm.

The sub-algorithm itself is created by invoking the `createSubAlgorithm()` method of the `Algorithm` base class. The parameters passed are the type of the algorithm, its name and a reference to the pointer which will be set to point to the newly created sub-algorithm. Note that the name passed into the `createSubAlgorithm()` method is the same name that should be used within the job options file for specifying algorithm properties.

```
Algorithm* m_pSubAlgorithm;   // Pointer to the sub algorithm
                              // Must be a member variable of the parent class
std::string type;             // Type of sub algorithm
std::string name;             // Name to be given to subAlgorithm
StatusCode sc;                // Status code returned by the call
sc = createSubAlgorithm(type, name, Algorithm*& m_pSubAlgorithm);
```

The algorithm type (i.e. class name) string is used by the application manager to decide which factory should create the algorithm object.

The execution of the sub-algorithm is entirely the responsibility of the parent algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework as shown in Figure 6. Similarly the properties of a sub-algorithm are also automatically set by the framework.

Note that the `createSubAlgorithm()` method returns a pointer to an `Algorithm` object not an `IAlgorithm` interface. This means that you have access to the methods of both the `IAlgorithm` and `IProperty` interfaces, and consequently as well as being able to call `execute()` etc. you can also explicitly call the `setProperty(Property&)` method of the sub-algorithm, as is done in the following code fragment. For this reason with nested algorithms you are not restricted to calling `setProperty()` only at initialisation. You may also change the properties of a sub-algorithm during the main event loop.

```
Algorithm *m_pSubAlgorithm;
sc = createSubAlgorithm(type, name, Algorithm*& m_pSubAlgorithm);
IntegerProperty p(“Counter”, 1024);
m_pSubAlgorithm->setProperty(p);
```

Note also that the vector of pointers to the sub-algorithms is available via the `subAlgorithms()` method.

5.5 Algorithm paths and filters

A physics application may need to execute different sequences of algorithms depending on the physics signature. We refer to each such sequence as a *path*. Within different paths we may want to execute the same algorithm class, but perhaps as a different *instance* with different configuration parameters. We may wish to terminate processing (to save CPU time) on a given path if a *filter* algorithm on the path fails some selection criteria.

A `Sequencer` class is available in the `GaudiAlg` package which manages algorithm paths using the filtering protocols which are implemented in the `Algorithm` class. Listing 7 is an extract of the job options file of the `AlgSequencer` example: a `Sequencer` instance is created (line 2) with two *members* (line 5); each member is itself a `Sequencer`, implementing the *paths* set up in lines 7 and 8, which consist of `Prescaler`, `EventCounter` and `HelloWorld` algorithms.

Algorithms can call `setFilterPassed( true/false )`. Algorithms downstream of one that sets this flag to `false` will not be executed, unless the `StopOverride` property of the Sequencer has been set (line 6).

An algorithm *instance* is executed only once per event, even if it appears in multiple Sequencers. Finally, algorithms can be enabled or disbaled at run time (which will become useful in an interactive scripting environment when one is implemented in Gaudi)

Listing 7 Example jobOptions for a sequencer algorithms

```
 1:  ApplicationMgr.DLLs   += { "GaudiAlg" };
 2:  ApplicationMgr.TopAlg = { "Sequencer/TopSequence" };
 3:
 4:  // Setup the next level sequencers and their members
 5:  TopSequence.Members = {"Sequencer/Sequence1", "Sequencer/Sequence2"};
 6:  TopSequence.StopOverride = true;
 7:  Sequence1.Members = {"Prescaler/Prescaler1", "HelloWorld",
     "EventCounter/Counter1"};
 8:  Sequence2.Members = {"Prescaler/Prescaler2", "HelloWorld",
     "EventCounter/Counter2"};
 9:
10:  Prescaler1.PercentPass = 50.;
11:  Prescaler2.PercentPass = 10.;
12:  Prescaler1.OutputLevel = 4;
13:  Prescaler2.OutputLevel = 4;
```

The `Prescaler` and `EventCounter` classes are general purpose algorithms distributed with the `GaudiAlg` package.

5.6 Algorithms vs. Services vs. Tools

Algorithms make use of framework services to perform their work. ***Services*** are generally sizeable components that are setup and initialized once at the beginning of the job by the framework and used by many algorithms as often as they are needed. It is not desirable in general to require more than one instance of each service. Services cannot have a “state” because there are many potential users of them so it would not be possible to guarantee that the state is preserved in between calls. Since Algorithms are called once per event by the framework, it is difficult to encapsulate a piece of code that needs to be called many times during the processing of a single event. Examples are: single track fitter algorithm, association to truth information algorithm, etc. For that we need to introduce another category of processing objects that can encapsulate these somehow more simple algorithms. We have called this category ***Tools***. Algorithms use their own private set of instances of Tools to perform their work. These instances are configured according to the needs of each particular Algorithm. If an Algorithm does not need to specially configure a Tool it can share the Tool instance with other Algorithms. A specific Service, the `ToolSvc`, acts as Tool provider (a kind of tool box or tool factory) to Algorithms. An Algorithm requests the tools it needs to `ToolSvc` and uses the provided abstract interfaces without having to know anything about the implementation. Tools and the ToolSvc are described in detail in Chapter 12

Chapter 6
Accessing data

6.1 Overview

The data stores are a key component in the application framework. All data which comes from persistent storage, or which is transferred between algorithms, or which is to be made persistent must reside within a data store. In this chapter we look at how to access data within the stores, and also at the `DataObject` base class and some classes related to it.

We also cover how to define your own data types and the steps necessary to save newly created objects to disk files. The writing of the converters necessary for the latter is covered in chapter 13.

6.2 Using the data stores

There are four data stores currently implemented within the Gaudi framework: the event data store, the detector data store, the histogram store and the n-tuple store. They are described in chapters 7, 8, 9 and 10 respectively. The stores themselves are no more than logical constructs with the actual access to the data being via the corresponding services. Both the event data service and the detector data service implement the same `IDataProviderSvc` interface, which can be used by algorithms to retrieve and store data. The histogram and n-tuple services implement extended versions of this interface (`IHistogramSvc, INTupleSvc`) which offer methods for creating and manipulating histograms and n-tuples, in addition to the data access methods provided by the other two stores.

Only objects of a type derived from the `DataObject` base class may be placed directly within a data store. Within the store the objects are arranged in a tree structure, just like a Unix file system. As an example consider Figure 7 which shows a part of the LHCb transient data model. An object is identified by its position in the tree expressed as a string such as: "`/Event`", or "`/Event/MC/MCParticles`". In principle the structure of the tree, i.e. the set

of all valid paths, may be deduced at run time by making repeated queries to the event data service, but this is unlikely to be useful in general since the structure will be largely fixed.

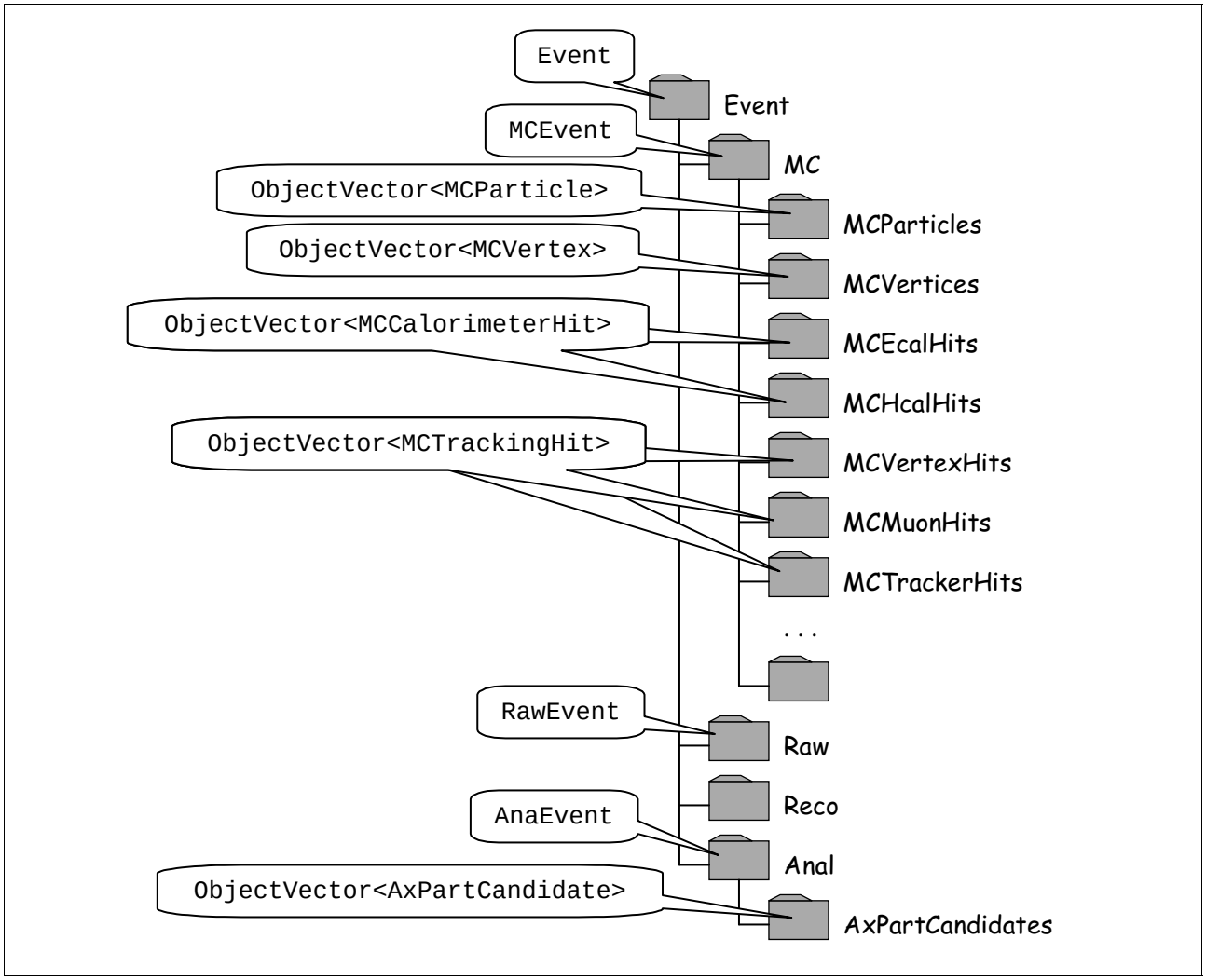


Figure 7 The structure of a part of the LHCb event data model.

As stated above all interactions between the data stores and algorithms should be via the `IDataProviderSvc` interface. The key methods for this interface are shown in Listing 8 but the API reference should be consulted for the complete version.

Listing 8 Some of the key methods of the `IDataProviderSvc` interface.

```
StatusCode findObject(const std::string& path, DataObject*& pObject);
StatusCode findObject(DataObject* node, const std::string& path,
                      DataObject*& pObject);
StatusCode retrieveObject(const std::string& path, DataObject*& pObject);
StatusCode retrieveObject(DataObject* node, const std::string& path,
                          DataObject*& pObject);

StatusCode registerObject(const std::string path, DataObject*& pObject);
StatusCode registerObject(DataObject *node, DataObject*& pObject);
```

The first four methods are for retrieving a pointer to an object that is already in the store. How the object got into the store, whether it has been read in from a persistent store or added to the store by an algorithm, is irrelevant.

The find and retrieve methods come in two versions: One version uses a full path name as an object identifier, the other takes a pointer to a previously retrieved object and the name of the object to look for below that node in the tree.

Additionally the “find” and “retrieve” methods differ in one important respect: the “find” method will look in the store to see if the object is present (i.e. in memory) and if it is not will return a null pointer. The “retrieve” method, however, will attempt to load the object from a persistent store (database or file) if it is not found in memory. Only if it is not found in the persistent data store will the method return a null pointer (and a bad status code of course).

6.3 Using data objects

Whatever the concrete type of the object you have retrieved from the store the pointer which you have is a pointer to a `DataObject`, so before you can do anything useful with that object you must cast it to the correct type, for example:

```
DataObject *pObject;
StatusCode sc = eventSvc()->retrieveObject(“/Event/MC/MCParticles”,pObject);
if( sc.isFailure() )
  return sc;

MCParticleVector *tv = 0;
try {
  tv = dynamic_cast<MCParticleVector *> (pObject);
} catch(...) {
  // Print out an error message and exit
}
// tv may now be manipulated.
```

where after the dynamic cast all of the methods of the `MCParticleVector` class become available. In the event that the object which is returned from the store does not match the type to which you try to cast it, an exception will be thrown. If you do not catch this exception then your program will exit, probably with an obscure message.

As mentioned earlier a certain amount of run-time investigation may be done into what data is available in the store. For example, suppose that we have various sets of testbeam data and each data set was taken with a different number of detectors. If the raw data is saved on a per-detector basis the number of sets will vary. The following code fragment in Listing 9 illustrates how an algorithm may loop over the data sets without knowing a priori how many there are.

Listing 9 Code fragment for accessing an object from the store

```
 1:  std::string objectPath = “Event/RawData”;
 2:  DataObject* pObject;
 3:  StatusCode sc;
 4:
 5:  sc = eventSvc()->retrieveObject(objectPath, pObject);
 6:
 7:  IdataDirectory *dir = pObject->directory();
 8:  IdataDirectory::DirIterator it;
 9:  for(it = dir->begin(); it != dir->end(); it++) {
10:
11:    DataObject *pDo;
12:    sc = retrieveObject(pObject, (*it)->localPath(), pDo);
13:
14:    // Do something with pDo
15:  }
```

The last two methods shown in Listing 8 are for registering objects into the store. Suppose that an algorithm creates objects of type UDO from, say, objects of type `MCParticle` and wishes to place these into the store for use by other algorithms. Code to do this might look something like:

Listing 10 Registering of objects into the event data store

```
 1:  UDO* pO; // Pointer to an object of type UDO (derived from DataObject)
 2:  StatusCode sc;
 3:
 4:  pO = new UDO;
 5:  sc = eventSvc()->registerObject(“/Event/Recon/tmp”,”OK”, pO);
 6:
 7:  // THE NEXT LINE IS AN ERROR, THE OBJECT NOW BELONGS TO THE STORE
 8:  delete pO;
 9:
10:  UDO autopO;
11:  // ERROR: AUTOMATIC OBJECTS MAY NOT BE REGISTERED
12:  sc = eventSvc()->registerObject(“/Event/Recon/tmp”, “notOK”, autopO);
```

Once an object is registered into the store, the algorithm which created it relinquishes ownership. In other words the object should not be `deleted`. This is also true for objects which are contained within other objects, such as those derived from or instantiated from the `ObjectVector` class (see the following section). Furthermore objects which are to be registered into the store must be created on the heap, i.e. they must be created with the `new` operator.

6.4 Object containers

As mentioned before, all objects which can be placed directly within one of the stores must be derived from the `DataObject` class. There is, however, another (indirect) way to store objects

within a store. This is by putting a set of objects (themselves not derived from DataObject and thus not directly storable) into an object which is derived from `DataObject` and which may thus be registered into a store.

An object container base class is implemented within the framework and a number of templated object container classes may be implemented in the future. For the moment, two "concrete" container classes are implemented: `ObjectVector<T>` and `ObjectList<T>`. These classes are based upon the STL classes and provide mostly the same interface. Unlike the STL containers which are essentially designed to hold objects, the container classes within the framework contain only pointers to objects, thus avoiding a lot of memory to memory copying.

A further difference with the STL containers is that the type `T` cannot be anything you like. It must be a type derived from the `ContainedObject` base class, see Figure 8. In this way all "contained" objects have a pointer back to their containing object. This is required, in particular, by the converters for dealing with links between objects. A ramification of this is that container objects may not contain other container objects (without the use of multiple inheritance).

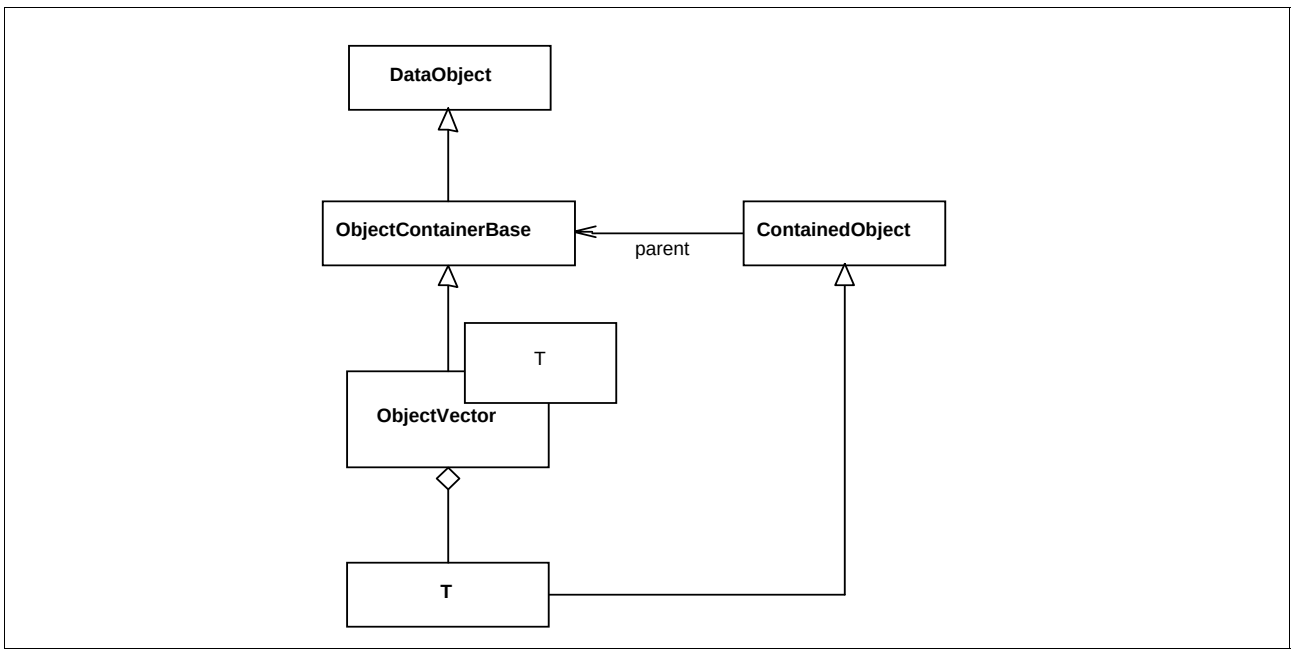


Figure 8 The relationship between the DataObject, ObjectVector and ContainedObject classes.

As mentioned above, objects which are contained within one of these container objects may not be located, or registered, individually within the store. Only the container object may be located via a call to `findObject()` or `retrieveObject()`. Thus with regard to interaction with the data stores a container object and the objects that it contains behave as a single object.

The intention is that "small" objects such as clusters, hits, tracks, etc. are derived from the `ContainedObject` base class and that in general algorithms will take object containers as their input data and produce new object containers of a different type as their output.

The reason behind this is essentially one of optimization. If all objects were treated on an equal footing, then there would be many more accesses to the persistent store to retrieve very

small objects. By grouping objects together like this we are able to have fewer accesses, with each access retrieving bigger objects.

6.5 Using object containers

The code fragment below shows the creation of an object container. This container can contain pointers to objects of type `MCTrackingHit` and only to objects of this type (including derived types). An object of the required type is created on the heap (i.e. via a call to `new`) and is added to the container with the standard STL call.

```
ObjectVector <MCTrackingHit>  hitContainer;
MCTrackingHit*   h1 = new MCTrackingHit;
hitContainer.push_back(h1);
```

After the call to `push_back()` the hit object “belongs” to the container. If the container is registered into the store, the hits that it contains will go with it. Note in particular that if you delete the container you will also delete its contents, i.e. all of the objects pointed to by the pointers in the container.

Removing an object from a container may be done in two semantically different ways. The difference being whether on removal from a container the object is also deleted or not. Removal with deletion may be achieved in several ways (following previous code fragment):

```
hitContainer.pop_back();
hitContainer.erase( end() );
delete h1;
```

The method `pop_back()` removes the last element in the container, whereas `erase()` maybe used to remove any other element via an iterator. In the code fragment above it is used to remove the last element also.

Deleting a contained object, the third option above, will automatically trigger its removal from the container. This is done by the destructor of the `ContainedObject` base class.

If you wish to remove an object from the container without destroying it (the second possible semantic) use the `release()` method:

```
hitContainer.release(h1);
```

Since the fate of a contained object is so closely tied to that of its container life would become more complex if objects could belong to more than one container. Suppose that an object belonged to two containers, one of which was deleted. Should the object be deleted and

removed from the second container, or not deleted? To avoid such issues an object is allowed to belong to a single container only.

If you wish to move an object from one container to another, you must first remove it from one and then add to the other. However, the first operation is done implicitly for you when you try to add an object to a second container:

```
container1.push_back(h1); // Add to fist container

container2.push_back(h1); // Move to second container
                          // Internally invokes release().
```

Since the object `h1` has a link back to its container, the `push_back()` method is able to first follow this link and invoke the `release()` method to remove the object from the first container, before adding it into the second.

In general your first exposure to object containers is likely to be when retrieving data from the event data store. The sample code in Listing 11 shows how, once you have retrieved an object container from the store you may iterate over its contents, just as with an STL vector. Note that the `typedef` is simply to save typing!

Listing 11 Use of the ObjectVector templated class.

```
 1:  typedef ObjectVector<MCParticle> MCParticleVector;
 2:  MCParticleVector *tracks;
 3:  MCParticleVector::iterator it;
 4:
 5:  for( it = tracks->begin(); it != tracks->end(); it++ ) {
 6:    // Get the energy of the track and histogram it
 7:    double energy = (*it)->fourMomentum().e();
 8:    m_hEnergyDist->fill( energy, 1. );
 9:  }
```

The variable `tracks` is set to point to an object in the event data store of type: `ObjectVector<MCParticle>` with a dynamic cast (not shown above). An iterator (i.e. a pointer like object for looping over the contents of the container) is defined on line 3 and this is used within the loop to point consecutively to each of the contained objects. In this case the objects contained within the `ObjectVector` are of type “pointer to `MCParticle`”. The iterator returns each object in turn and in the example, the energy of the object is used to fill a histogram.

6.6 Data access checklist

A little reminder:

- Do not `delete` objects that you have registered.
- Do not `delete` objects that are contained within an object that you have registered.

- Do not register local objects, i.e. objects NOT created with the `new` operator.
- Do not delete objects which you got from the store via `findObject()` or `retrieveObject()`.
- Do delete objects which you create on the heap, i.e. by a call to `new`, and which you do not register into a store.

6.7 Defining new data types

Most of the data types which will be used within Gaudi will be used by everybody and thus packaged and documented centrally. However, for your own private development work you may wish to create objects of your own types which of course you can always do with C++ (or Java) . However, if you wish to place these objects within a store, either so as to pass them between algorithms or to have them later saved into a database or file, then you must derive your type from either the `DataObject` or `ContainedObject` base class.

Consider the example below:

```
const static CLID CLID_UDO = 135; // Collaboration wide Unique number

class UDO : public DataObject {
public:
  UDO() : DataObject(), m_n(0) {
  }

  static CLID& classID() { return CLID_UDO; }
  virtual CLID& clID() { return CLID_UDO; }

  int n(){ return m_n; }
  void setN(int n){ m_n = n; }

private:
  int m_n;
}
```

This defines a class `UDO` which since it derives from `DataObject` may be registered into, say, the event data store. (The class itself is not very useful as its sole attribute is a single integer and it has no behaviour).

The thing to note here is that if the appropriate converter is supplied, as discussed in Chapter 13, then this class may also be saved into a persistent store (e.g. a ROOT file or an Objectivity database) and read back at a later date. In order for the persistency to work two things are required: the unique class identifier number (`CLID_UDO` in the example), and the `clID()` method which returns this identifier.

Types which are derived from `ContainedObject` are implemented in the same way. The only point to notice is that it is the `classID()` method which must be implemented. This is because contained objects may only reside in the store when they belong to a container, e.g. an

ObjectVector<T> which is registered into the store. The class identifier of a concrete object container class is calculated (at run time) from the type of the objects which it contains. Since the container may be empty a static method is required.

6.8 The SmartDataPtr/SmartDataLocator utilities

The usage of the data services is simple, but extensive status checking and other things tend to make the code difficult to read. It would be more convenient to access data items in the store in a similar way to accessing objects with a C++ pointer. This is achieved with smart pointers, which hide the internals of the data services.

6.8.1 Using SmartDataPtr/SmartDataLocator objects

The SmartDataPtr and a SmartDataLocator are smart pointers that differ by the access to the data store. SmartDataPtr first checks whether the requested object is present in the transient store and loads it if necessary. SmartDataLocator only checks for the presence of the object but does not attempt to load it.

Both SmartDataPtr and SmartDataLocator objects use the data service to get hold of the requested object and deliver it to the user. Since both objects have similar behaviour and the same user interface, in the following only the SmartDataPtr is discussed.

An example use of the SmartDataPtr class is shown below.

Listing 12 Use of a SmartDataPtr object.

```
 1:  StatusCode myAlgo::execute() {
 2:    MsgStream log(msgSvc(), name());
 3:    SmartDataPtr<Event> evt(eventSvc(),"/Event");
 4:    if ( evt ) {
 5:      // Print the event number
 6:      log << MSG::INFO << " Run:" << evt->run()
 7:          << " Event:" << evt->event() << endreq;
 8:    }
 9:    else {
10:      log << MSG::ERROR << "Error accessing event" << endreq;
11:      return StatusCode::FAILURE;
12:    }
13:  }
```

The SmartDataPtr class can be thought of as a normal C++ pointer having a constructor. It is used in the same way as a normal C++ pointer.

The SmartDataPtr and SmartDataLocator offer a number of possible constructors and operators to cover a wide range of needs when accesing data stores. Check the online reference documentation [2] for up-to date information concerning the interface of these utilities.

6.9 Smart references and Smart reference vectors

Smart references and Smart reference vectors are similar to smart pointers, they are used within data objects to reference other objects in the transient data store. They provide safe data access and automate the loading on demand of referenced data, and should be used instead of C++ pointers. For example, suppose that MC particles are already loaded but MC vertices are not, and that an algorithm dereferences a variable pointing to the origin vertex: if a smart reference is used, the MC vertices would be loaded automatically and only after that would the variable be dereferenced. If a C++ plain pointer were used instead, the program would crash. Smart references provide an automatic conversion to a pointer to the object and load the object from the persistent medium during the conversion process.

Smart references and Smart reference vectors are declared inside a class as:

```
class MCParticle {
  private:
    /// Smart reference to origin vertex
    SmartRef<MCVertex>        m_originMCVertex;
    /// Vector of smart references to decay vertices
    SmartRefVector<MCVertex>  m_decayMCVertices;
  public:
    /// Access the origin Vertex
    /// Note: When the smart reference is converted to MCVertex* the object
    /// will be loaded from the persistent medium.
  MCVertex* originMCVertex()   { return m_originMCVertex; }
}
```

The syntax of usage of smart references is identical to plain C++ pointers. The Algorithm only sees a pointer to the MCVertex object:

```
// Use a SmartDataPtr to get the MC particles from the event store
SmartDataPtr<MCParticleVector> particles(eventSvc(),"/Event/MC/MCParticles");
MCParticleVector::const_iterator iter;

// Loop over the particles to access the MCVertex via the SmartRef
for( iter = particles->begin(); iter != particles->end(); iter++ ) {
  MCVertex* originVtx = (*iter)->originMCVertex();
  if( 0 != originVtx ) {
    std::cout << "Origin vertex = " << *(*iter) << std::endl; }
}
```

All LHCbEvent data types use the Smart references and Smart reference vectors to reference themselves.

6.10 Saving data to a persistent store

Suppose that you have defined your own data type as discussed in the previous section. Suppose futhermore that you have an algorithm which uses, say, SicB data to create instances of your object type which you then register into the transient event store. How can you save these objects for use at a later date?

You must do the following:

- Write the appropriate converter (see Chapter 13)
- Put some instructions (i.e. options) into the job option file

Register your object in the store us usual, typically in the `execute()` method of your algorithm.

```
// myAlg implementation file

StatusCode myAlg::execute() {
  // Create a UDO object and register it into the event data store
  UDO* p = new UDO();
  eventSvc->registerObject("/Event/myStuff/myUDO", p);
}
```

In order to actually trigger the conversion and saving of the objects at the end of the current event processing it is necessary to inform the application manager. This requires some options to be specified in the job options file:

```
ApplicationMgr.OutStream  = { "DstWriter" };

DstWriter.ItemList         = { "/Event#1", "/Event/MyTracks#1"};
DstWriter.EvtDataSvc       = "EventDataSvc";
DstWriter.EvtConversionSvc = "RootEventCnvSvc";
DstWriter.OutputFile       = "result.root";
```

The first option tells the application manager that you wish to create an output stream called "DSTWriter". You may create as many output streams as you like and give them whatever name you prefer.

For each output stream object which you create you must set several properties. The `ItemList` option specifies the list of paths to the objects which you wish to write to this output stream. The number after the "#" symbol denotes the number of directory levels below the specified path which should be traversed. The `EvtDataSvc` option specifies in which transient data service the output stream should search for the objects in the `ItemList`. The `EvtConversionSvc` option specifies the conversion service which should be used to convert the objects and the output file or database name is set with the `OutputFile` option.

In addition to this the event persistency service must be set up correctly by specifying all necessary conversion services in the job options as shown in the much more comprehensive example distributed with the framework (`Rio.Example1`).

Chapter 7
LHCb Event Data Model

In this chapter we present the structure of the LHCb Event Data model. For the moment only a small part of the model is implemented in the public release. Various sub-detector specific parts are under development and are implemented in private code. The table at *http://lhcb.cern.ch/computing/Support/html/ConvertedSICBBanks.htm* shows the current implementation status by comparison with SICB banks.

7.1 Top level event data structures

The event data objects are located in the event data store, which is one of the stored described in Chapter 6. The data in the store are arranged in a tree. This facilitates the location of objects within the store by human-readable identifiers. The tree structure consists of an Event root branching four sub-trees: Monte Carlo event, Raw event, Reconstructed event, and Analysis event. A fifth branch for FrontEnd (FE) has been implemented in private code (available under L1/VELO CVS tree). Retrieving an identifiable object (see Chapter 6) from the store is based on the knowledge of the logical path. The paths so far implented are shown in the tables that follow. They are defined in the files:

`LHCbEvent/TopLevel/EventModel.h`

`LHCbEvent/TopLevel/EventModel.cpp:`

Table 4 Top level Event data model

Logical Path	Type
"/Event"	Event
"/Event/MC"	MCEvent
"/Event/Raw"	RawEvent
"/Event/Rec"	RecEvent
"/Event/Anal"	AnalEvent

The `LHCbEvent/TopLevel` directory also contains the include files for the top level event classes (which are all derived from `DataObject` and are therefore identifiable). The identifiable objects in the MonteCarlo, Raw, Reconstructed, and Analysis event sub-trees are all container classes, containing instances of classes that inherit (directly or indirectly) from the class `ContainedObject`.

7.2 Monte Carlo event

The Monte Carlo event sub-tree contains output from the event generators and from the Monte Carlo tracking. The include files for the contained classes can be found in `LHCbEvent/MonteCarlo/*.h` with obvious names

Table 5 Monte Carlo Event data model

Logical Path	Container Type
"/Event/MC/MCParticles"	MCParticleVector
"/Event/MC/MCVertices"	MCVertexVector
"/Event/MC/MCTrackerHits"	MCTrackingHitVector
"/Event/MC/MCVertexHits"	MCTrackingHitVector
"/Event/MC/MCVertexPileUpHits"	MCTrackingHitVector
"/Event/MC/MCMuonHits"	MCTrackingHitVector
"/Event/MC/MCRichRadiatorHits"	MCRichRadiatorHitVector
"/Event/MC/MCRichPhotodetectorHits"	MCRichPhotodetectorHitVector
"/Event/MC/MCECalHits"	MCCalorimeterHitVector
"/Event/MC/MCHCalHits"	MCCalorimeterHitVector
"/Event/MC/MCPreshowerHits"	MCCalorimeterHitVector

7.3 Raw event

The Raw event sub-tree should contain the raw data collected by the data acquisition and simulated data in the same format (i.e. with detector and electronics response applied). The include files for the contained classes can be found in `LHCbEvent/Raw/*.h` with obvious names. Currently only the classes `RawInnerTrackerMeass` and `RawOuterTrackerMeass` are implemented, which are for the time being copies of the SicB banks WIDG and WODG

Table 6 Raw Event data model

Logical Path	Container Type
"/Event/Raw/RawOuterTrackerMeas"	RawOuterTrackerMeasVector
"/Event/Raw/RawInnerTrackerMeas"	RawInnerTrackerMeasVector

7.4 Reconstructed event

The Reconstructed event sub-tree is meant to contain the output of the reconstruction program. It is currently empty.

7.5 Analysis event

The Analysis event sub-tree should contain the objects created and used during data analysis.The include files for the contained classes can be found in `LHCbEvent/Analysis/*.h` with obvious names. Currently only the class `AxPartCandidate` is implemented, derived from the SICB bank AXTK.

Table 7 Analysis Event data model

Logical Path	Container Type
"/Event/Anal/AxPartCandidates"	AxPartCandidateVector

7.6 Utilities

A series of utility classes are defined:

- `DetectorDataObject`, `CLHEPStreams`, `TimeStamp`, `Classification`, `ProcessingVersion`, `CellId`, `ParticleId`, `TriggerPattern`, `RandomNumberSeed`, `SummaryPID`, and `SummarySeesDetectors`. These are used in several places in the LHCb event data model.

Chapter 8
Detector Description

8.1 Overview

In this chapter we describe how we make available to the physics application developed using the framework the information related to the detector that resides in the detector description database (DDDB). The DDDB is the persistent storage for all the versions of the detector data needed to describe and qualify the detecting apparatus in order to interpret the event data.

The final clients of the detector description are the algorithms that need this information in order to perform their job (reconstruction, simulation, etc.). To provide this information, there needs to be a sub-detector specific part that understands the sub-detector in question and uses a set of common services. The detector description we are providing in GAUDI is nothing else than a framework for developers to provide the specific detector information to algorithms by using as much as possible common or generic services.

In the following sections we begin with an overview of the DDDB. We then discuss how to access the detector description data in the Gaudi transient detector data store. This is followed by a discussion of the logical structure of the Gaudi detector description. Finally we describe in detail how the detector decription can be built and made persistent using the XML markup language.

8.2 Detector Description Database

The detector description database (DDDB), see Figure 9, includes a physical and a logical description of the detector. The physical description covers dimensions, shape and material of the different types of elements from which the detector is constructed. There is also information which corresponds to each element which is actually manufactured and assembled into a detector, for example the positioning of each element. Both active and passive elements should be included. The description of active elements should allow for the

specification of deficiencies (dead channels), alignment corrections, etc. and also detector response characteristics, e.g. energy normalization in calorimeters, drift velocity in gas chambers.

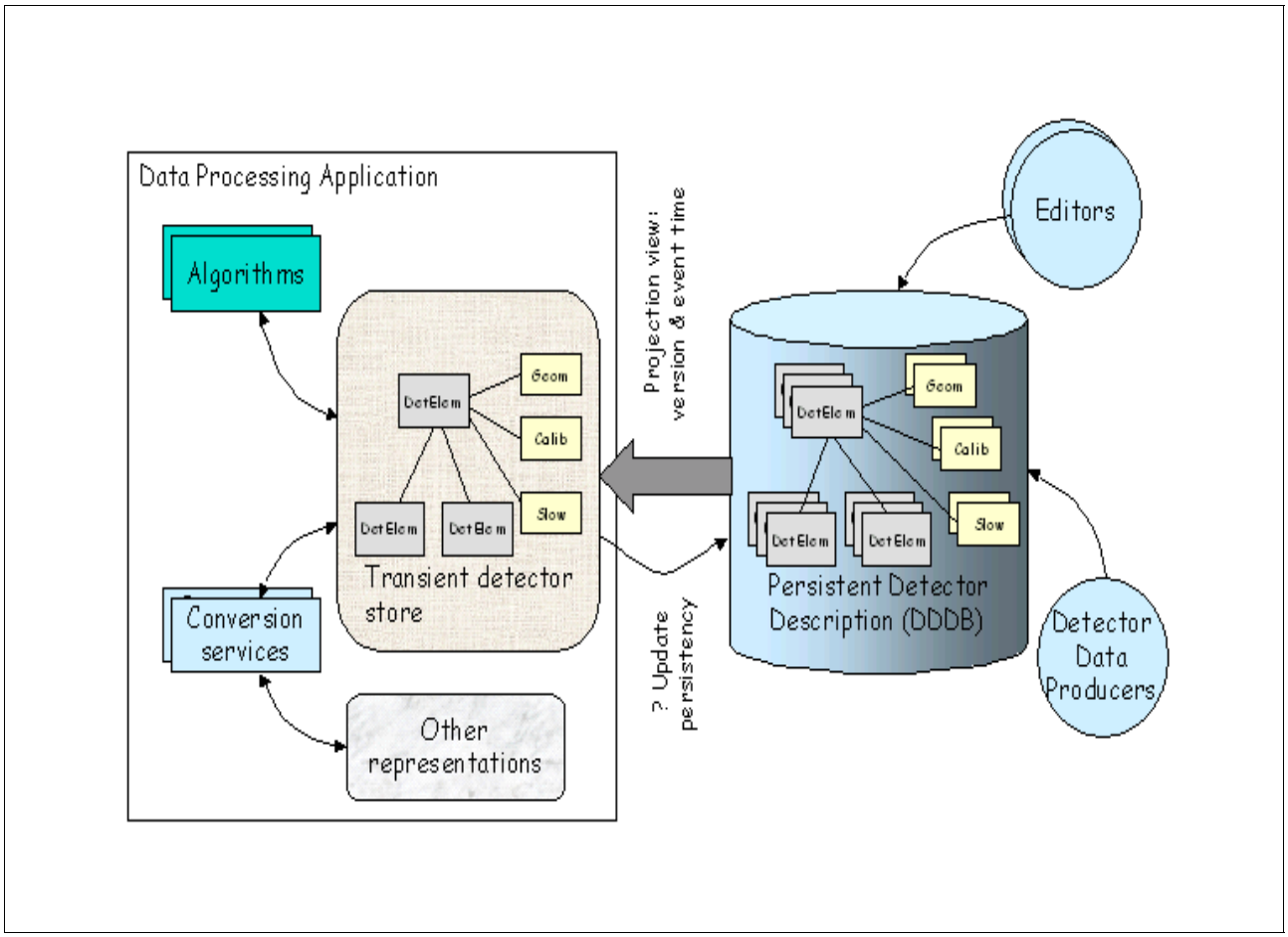


Figure 9 Overview of the Detector Description model.

The logical description provides two main functions. The first is a simplified access to particular parts of a physical detector description. This could be a hierarchical description where the a given detector setup is composed of various sub-detectors, each of which is made up of a number stations, modules or layers, etc. and there would be a simple way for a client to use this description to navigate to the information of interest. The second function of the logical description is to provide a means of detector element identification. This allows for different sets of information which are correlated to specific detector elements to be correctly associated with each other

In a detector description, the definition of the detector elements and the data associated to their physical description may vary over time, for instance due to real or hypothetical changes to the detector. Each such change should be recorded as a different version of the detector element. Additionally, it should be possible to capture for an entire description a version of each of the elements and associate a name to that set. This is similar to the way CVS allows one to tag a set of files so that one does not need to know the independent version numbers for each file in the set.

8.3 Using the Detector Data transient store

8.3.1 Structure of the transient store

The transient representation of the detector description is implemented as a standard GAUDI data store. Refer to Chapter 6 for general information on how to access data in stores. The structure of the detector transient store is represented in Figure 10. There are various branches in the hierarchical tree structure and probably there will be more of them defined in the future.

At present there are three top level catalogs in the transient store. The main catalog, called “Structure” contains the logical structure of the detector identified by the “setup name” i.e. “LHCb” containing the description of the detector and this catalog is used for identification and navigation purposes. Other catalogs are the palette of logical volumes and solids, called “Geometry”, used for the geometry description and the palette of materials, called “Materials”, used to describe the material properties of the solids needed for the detector simulation, etc.

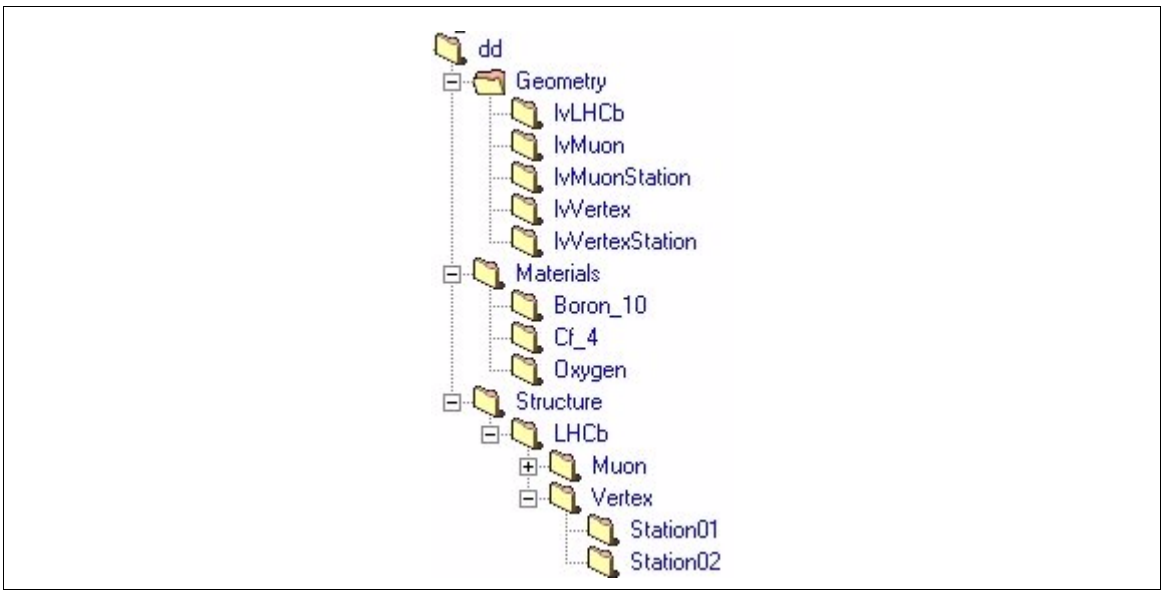


Figure 10 The structure of part of the LHCb detector data transient store.

8.3.2 Accessing detector data on the transient store

An algorithm that needs to access a given detector part uses the detector data service to locate the relevant `DetectorElement`. This operation of locating the required detector description object can be generally done during the initialization phase of the algorithm. Contrary to the Event Data, the Detector Data store is not cleared for each event and the references to detector elements remain valid and are updated automatically during the execution of the program.

Locating the relevant detector element is done using the standard `IDataProviderSvc` interface via the `detSvc()` accessor as shown in the code fragment below:.

Listing 13 Retrieving a detector element by using smart pointers

```
SmartDataPtr<DetectorElement> vertex( detSvc(),
                                      "/dd/Structure/Vertex/VStation01");
if( !vertex ) {
  // Error, detector element has not been retrieved
}
```

Similarly the user can retrieve an array of such references. The following code fragment can be used to prepare an array with pointers to all of the Muon stations. Here we use an STL vector of pointers to `DeMuonStation` objects to store the retrieved Muon stations.

Listing 14 Retrieving a vector of detector elements using smart references

```
std::vector<DeMuonStation*> d_stations;

SmartDataPtr<DetectorElement> stations(detSvc(),
                                       "/dd/Structure/LHCb/Muon/Stations" );
if( !stations ) {
  return StatusCode::FAILURE;
}

/// Loop over all the muon stations found in the detector model
for ( DataObject::DirIterator d =  stations->dirBegin();
                              d != stations->dirEnd();
                                d++ )
{
  SmartDataPtr<DeMuonStation> s( detSvc(),(*d)->fullpath() );
  if( !s ) {
    return sc;
  }
  d_stations.push_back( s );
}
```

8.3.3 Using the DetectorElement class

The `DetectorElement` class implements the `IDetectorElement` interface. Currently, only the `geometry()` accessor method is implemented. The rest of the interface will be implemented in the next releases of the detector description package. In addition, `DetectorElement` implements the `IValidity` interface. This interface is used to check if the detector element is synchronized with the current event. If the detector element contains information no longer valid at the time the current event was generated, its content must be updated from the persistent storage. In the current implementation it is not foreseen for end users to use this interface directly.

The accessor method `geometry()` gives access to geometry information offered by the interface of type `IGeometryInfo`. This interface allows the retrieval of a reference to a logical volume associated with the given detector element, its material property, the position in the geometrical hierarchy. In addition to that you can ask it questions like:

1. Transformation matrix from the Global to the Local Reference system
2. Transformation matrix from the Local to the Global Reference system
3. Perform transformation of point from the Global to Local Reference system
4. Perform transformation of point from the Local to Global Reference system
5. Name of daughter volume (of current volume) which contains given (global) point
6. Get a pointer to daughter volume which contains given (global) point
7. Name of daughter volume (at deeper hierarchical level) which contains given point
8. Get a pointer to daughter volume (on deeper level) to which contains given point
9. Get the exact full geometry location[1]
10. Whether the given point is inside the associated logical volume or not
11. A pointer to the associated logical volume

For example:

Listing 15 Getting pointer to a logical volume and retrieving its various properties

```
SmartDataPtr<DetectorElement> vs(detSvc(),
                                 "/dd/Structure/LHCb/Vertex/VStation01");
if( !vs ) {
  return StatusCode::FAILURE;
}

/// Report the material and its radiation length
ILVolume* stvol = vs->geometry()->lvolume();
log << MSG::INFO << vs->fullpath() << " is made of " << stvol->materialName()
    << " with radiation length " << stvol->material()->radiationLength()
    << endreq;

/// Retrieve the shape information
const ISolid*  stsolid = stvol->solid();

/// Get the rotation and translation
HepTransform3D sttrans = vs->geometry()->matrixInv();
HepRotation    strot   = sttrans.getRotation();
Hep3Vector     stvec   = sttrans.getTranslation();
```

8.4 General features of the geometry tree

The construction of the geometry tree within the Gaudi framework is based on the following postulates:

- The geometry tree is constructed from *Logical Volumes* and *Physical Volumes*.

1. This operation can be time consuming!

- There are no "up-links" in the geometry tree. It means that each node has no information about the "up" (or "parent", "mother") node.
- Each *Logical Volume* has a information about its "down" ("children") nodes, represented by *Physical Volumes*
- Each *Logical Volume* has information about its shape and dimensions ("*Solid*").
- Each *Logical Volume* has access to information about the material content.
- Neither *Logical Volumes* nor *Physical Volumes* have any information about their absolute position in the space.
- *Logical Volumes* have no information about their own position relative to other *Logical Volumes*.
- Each *Physical Volume* has a information about its position inside the mother ("parent") *Logical Volume*. This is the only geometry information available in the whole tree.
- All boolean operations on Logical Volumes and Physical Volumes are strictly forbidden[1]. Boolean operations should be performed at the level of *Solids*. This is one of the most essential postulates of the Gaudi geometry structure.

The geometry tree which fulfils all these postulates represents a very effective, simple and convenient tool for description of the geometry. Such a tree is easily formalized. It has many features which are similar to the features of the geometry tree used within the Geant4 toolkit and could easily be transformed to the Geant4 geometry description.

Some consequences of these postulates are:

- The top-level *Logical Volume* (presumably the experimental hall, or cave, or the whole LHCb detector) defines the absolute coordinate reference system. In other words, the null-point (0,0,0) in the so called *Global Reference System* is just the center of the top *Logical Volume*.
- All geometry calculations, computations, inputs and outputs, performed with the usage of a *Logical Volume* are in the local reference system of this *Logical Volume*.
- All geometry calculations, computations, inputs and outputs, performed with the usage of a *Physical Volume* are in the local reference system of its parent *Logical Volume*.

Sometimes one needs a more efficient way of extracting information from the geometry tree or to compute the unique location of a point in the geometry tree. For these purposes, a simplified detector description tree is introduced into the GAUDI framework[2].

The next subsections give brief details of the implementations of Logical Volumes, Physics Volumes and Solids in Gaudi. More detailed documentation can be found at *http://lhcb.cern.ch/computing/Components/html/GaudiReference.htm*

1. This is equivalent to the absence of the 'MANY' flag in the GEANT3 toolkit.
2. Within the Geant4 toolkit there exist two approaches for solving the same problem: *Read-Out-Geometry Tree* and *Navigator*. Our approach is quite close to the combined usage of both.

8.4.1 Logical Volumes

The notion of *Logical Volume* is implemented in GAUDI by the `class LVolume`. LVolume is an identifiable object and therefore inherits from `class DataObject` and can be identified in the transient data store by a unique name (its “path”). It implements the `ILVolume` and `IValidity` interfaces.

8.4.2 Physical Volumes

The notion of *Physical Volume* in the Gaudi geometry is extremely primitive, it is just a *Logical volume* which *is positioned* inside its mother *Logical Volume*. It consists of the name of the *Logical Volume* to be positioned inside the mother *Logical Volume*, together with the transformation matrix from the local reference system of the mother *Logical Volume* to the local reference system of the daughter *Logical Volume*. This is implemented in Gaudi by the `class PVolume`. PVolume is not identifiable and implements the `IPVolume` interface.

8.4.3 Solids

All solids implement the `ISolid` interface. Currently, five types of “primitive” solids are implemented: *Boxes, Trapezoids, Tube segments, Conical tube segments* and *Sphere segments*. These were chosen from the most frequently used shapes in the GEANT3 and GEANT4 toolkits - more shapes can be implemented if necessary. In addition, *Boolean Solids* have been defined, which allow *Subtraction, Union and Intersection* operations on solids, to build complex shapes.

8.5 Persistent representation

The GAUDI detector description is based on text files whose structure is described by XML (eXtendable Markup Language), a standard language which allows the definition of custom tags, unlike the fixed set of tags of HTML used for WWW. XML files are understandable by humans as well as computers. Data in XML are self-descriptive so that by looking at the XML data one can easily guess what the data mean. Unlike the HTML tags, tags in XML do not define how to render or visualize the data. This is left to an application which understands the data and can visualize them if wanted. An advantage of XML is that there exists plenty of software which can be used for parsing and analysing, as it is an industry standard.

In the future we expect to replace the text files by an object persistency based on an object oriented database, for example ObjectivityDB, but the data will continue to be described in XML.

8.5.1 Brief introduction to XML

8.5.1.1 XML Basics

XML is extendible, meaning that a user can define his own markup language with his own tags. The tags define the meaning of the data they represent or contain. They act as data describers. For example, Listing 16: shows the XML file which describes an e-mail.

Listing 16 Simple XML file describing an e-mail

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- This is an example of XML -->
<Email>
  <TimeStamp time="11:38:43" date="22/11/1999" />
  <Sender>sender@cern.ch</Sender>
  <Recipient>recipient@cern.ch</Recipient>
  <Subject>Lunch...</Subject>
  <Body> Could we meet at 14:00?
    <Signature>Sender's signature</Signature>
  </Body>
</Email>
```

At the first look this markup language looks like screwed-up HTML. This is because both HTML and XML have their roots in SGML, but they are used for different purposes. From the example above it's not clear how to present the data described there nor how to visualize them. What is clear however is the meaning of the data items encoded in XML. Thus one can easily recognize the data items and guess what they mean. On the other hand it is relatively easy to instruct a computer program what to do with the given data item according to the XML markup elements it is encapsulated in. Let us analyse the example shown above.

8.5.1.2 XML components

XML declaration must be at the beginning of each XML document. It is the first line in the example. It says that this file is an XML file conforming to the XML standard version 1.0 and is encoded in UTF-8 encoding. The encoding is very important because XML has been designed to describe data using the Unicode standard for text encoding. This means that all XML documents are treated as 16 bit Unicode characters instead of usual ASCII. So, even if you write your XML files using 7 or 8 bit ASCII, all the XML applications will work with it as with 16 bit Unicode XML data. The encoding information is important, for example when an XML document is transferred over the Internet to some other country where a different encoding is used. If the receiving application can guess the XML encoding from the received file, it can apply transcoding methods to convert the file into proper local encoding, thus preserving readability of the data.

XML comments look like comments in SGML or HTML. They start with **`<!--`** and end with **`-->`**. Comments in XML cannot be nested.

XML elements are the markup components describing data in XML. In the example we had the following XML elements: `Email, TimeStamp, Sender, Recipient, Subject, Body, Signature`. The very basic and mandatory rule of XML is that all XML element tags

must nest properly and there **must be only one** root XML element at the top level of each XML document, which contains all the others. Proper nesting means that each XML element has its opening and closing tag and the closing tag must appear before the parent element's closing tag, as shown in Listing 17. Following these rules one can always produce **well-formed** XML documents.

Listing 17 XML elements syntax and proper vs. invalid nesting

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<!-- Root tag is top level root element of XML file -->
<Root>
  <!-- Elements which are empty -->
  <EmptyElement />
  <EmptyWithAttributes attr1="first" attr2='second' />
  <!-- Elements having content model -->
  <ProperNesting> <Something>In</Something> </ProperNesting>
  <WRONGNESTING> <BADTHING>huhu </WRONGNESTING> </BADTHING>
</Root>
```

XML elements can have attributes and a content. Attributes usually describe the properties of the given element. The value of the attribute follows the assignment operator and is enclosed inside double or single quotes.In the content can appear text data or other properly nested elements. The text data and nested elements can be mixed inside the content.

8.5.1.3 Well formed versus Valid XML documents

A well formed XML document is any XML document which follows the rules described in the previous section. However this is not sufficient in some cases. Inside an XML document you can have any XML tag you can imagine. This is not harmful to XML itself but makes it impossible to process such a document with a computer program and makes it very hard to maintain the program to keep it synchronised with all the new tags users invent. For a well formed XML document is not possible to check or **validate** that the document contains only the tags which make sense or have valid content. For that purpose there exists a notation called **Document Type Definition** (DTD) which comes from SGML world. This notation permits the definition of a grammar (a valid set of tags, their contents and attributes) which allows then to perform the validation of XML documents, i.e. whether the document contains the XML tags which belong to the set of tags defined by its associated DTD. In addition, the validating application can check whether the tags contain only allowed tags or text data, and whether the tags use only attributes defined for them by the DTD. The validation process can can also perform **normalization of attributes,** i.e. assign default values to attributes that the user has omitted because they are described as optional or with a fixed value in the DTD.

Important note: the default behaviour of the validating application, recommended by the XML standard, is to stop parsing of an XML document in case of an error. This is because the XML files describe data and an error in XML means corrupted data.

8.5.2 Working with the persistent detector description in XML

In this section we describe how to create or update the XML data files used by the detector description database. We will go through all available XML elements defined for the detector description, explain their XML syntax and show how to use them. For convenience we have included pseudo UML diagrams showing content model definitions of the XML tags. This approach has been chosen because it is easier to understand the graphical form than raw DTD definitions. Here follows a brief explanation how to read these pseudo UML diagrams:

- Elements are represented as boxes containing their name and, optionally, the specification of their content (above the name, inside << >> brackets) which can be EMPTY or ANY.
- The directed arrows are used to describe the containment relation. The containing element has the diamond of an arrow attached to it and the contained element is at the pointed end of the arrow. A multiplicity (see Table 8) is specified at the end of the arrow to say how many of the elements can be placed inside the parent element. This is very important for validation purposes.

Table 8 Multiplicity used in the content model definitions

Diagram notation	DTD notation	Description
1		Exactly one
0..1	?	Optional, one or zero
1..*	+	One or more
0..*	*	Zero or more

- The **OR** keyword in the diagram means that the element may or may not contain the child elements connected to it by arrows. If the keyword is missing in the diagram then the sequence of the child elements (from left to right) is assumed and multiplicity can play important role in this context.
- Attributes of elements are shown as data members, with their type specified and optionally a valid or default values. The type can be any of the types shown in the Table 9. This is not the complete list of attribute types available, just those used by the detector description.

Table 9 Attribute types

Type	Description
ID	Unique identifier
IDREF	Reference to a unique identifier
ENUM	Enumeration type
CDATA	Character data (string)

- The DTD also specifies whether the attribute is optional, required or has fixed or default value. This information is not shown in the diagrams and the user has to look at the concrete DTD file in order to get this information.

8.5.2.1 Defining the top level structure

To fulfil the XML basic rule that each XML document must have only the one root XML element we have defined the element **DDDB**, see Figure 11.

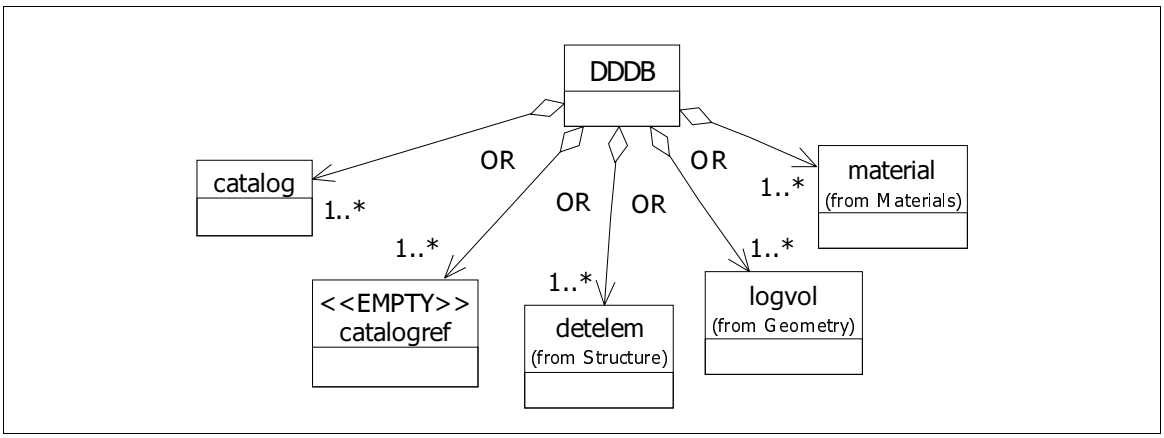


Figure 11 Content model definition of the root element for detector description

This element must be present in each of the XML files created for the detector description. This element is so defined that it can contain all the important elements needed for detector description and thus the database can be spread over multiple XML files. The **DDDB** element can contain one or more of the following elements: `catalog, catalogref, detelem, logvol, material`. Each of these elements will be described in detail later

The detector description in XML requires a bootstrap file which contains a definition of the database root and a definition of the top level catalogs mentioned above. This done in a flexible way to make it possible to have multiple bootstrap files. This can be done in the job options by setting the application manager properties `DetDbLocation` and `DetDbRootName`. They allow to switch from one bootstrap file to another and to switch between root nodes. The bootstrap file must have defined the root node catalog as the only child of the **DDDB** element. Inside the root node catalog the top level catalogs can be defined according to the required content. See the Listing 18 for an example.

Listing 18 Example of a boostrap file for the detector description

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <catalog classID="3" name="dd">
    <catalogref href="catalogs.xml#Structure" />
    <catalogref classID="3" href="catalogs.xml#Materials" />
    <catalog name="Geometry">
      <logvolref href="geometry.xml#lvLHCb" />
      <logvolref href="geometry.xml#lvVertex" />
      <logvolref href="geometry.xml#lvVStation" />
    </catalog>
  </catalog>
</DDDB>
```

In the example, the root nodes catalog `/dd` and its three top level catalogs for logical detector description, geometry and materials are defined. Note that some of the catalog and catalogref tags have not specified the `classID` attribute. This shows how the validation can reduce the amount of the text in XML because these values are automatically included after the document has been validated against the DTD file `xmldb.dtd` specified in the `DOCTYPE` section.

The catalog and catalogrefs are used as bookshelves or references to bookshelves respectively to achieve a possibility to split the XML database into logical partitions. The catalogref element acts as forward reference saying where to find the given catalog definition. This tag is one of the XML references defined for the detector description. All of them have almost the same usage and syntax. The common things they have are attributes for class ID and hyperlink reference. The difference is the value of the class ID of the objects they point to, also some of them may contain other elements. The hyperlink is in general specified using the format:

`protocol`[1]`://hostname/path/to/the/file.xml#ObjectID` or `#ObjectID`

The `protocol` and `hostname` parts can be omitted if the file resides on the local host. It is possible to write a hyperlink without the full path name in case one needs to refer to an XML object residing inside another file. In this case the relative path will be appended to the location of the currently parsed XML file.

For example having the current file location `/full/path/to/current.xml` and inside this file a hyperlink as `href="next/file.xml#NextOID"` the hyperlink will be resolved as `/full/path/to/next/file.xml#NextOID`.

If the hyperlink has the form `#ObjectID` this means that the referred object is located in the same file. For content model definitions of the catalog and catalogref elements see Figure 12

1. The current implementation supports only file:/ / protocol, this is due to limited support in the XML4C library from IBM

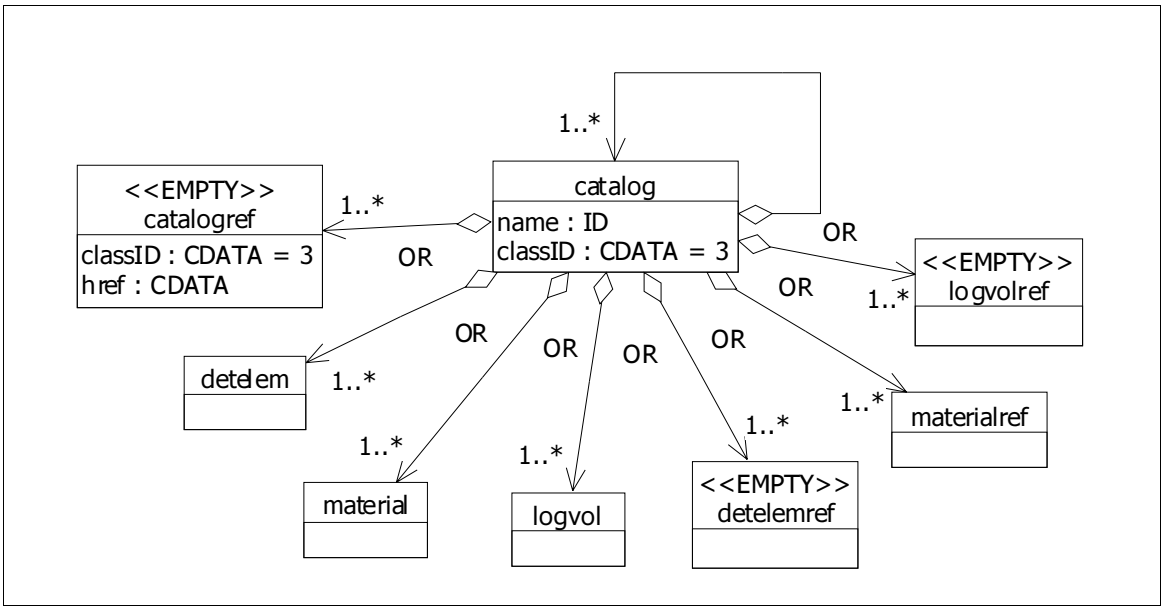


Figure 12 Content model definition of the catalog and catalogref elements

8.5.2.2 Defining detector elements

Figure 13 shows the content model definition of the `detelem` element and its child elements.

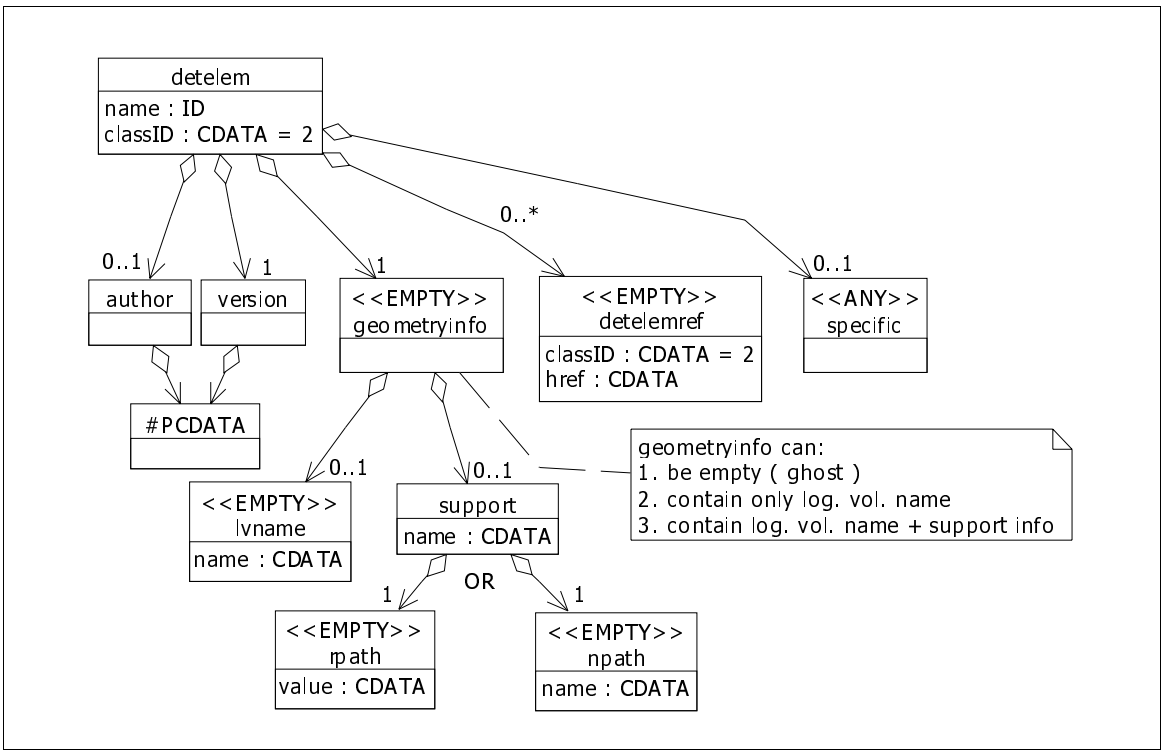


Figure 13 Content model definition of the detelem element

The detelem element has a sequence of tags which must appear in the same order shown on the diagram (author, version, geometryinfo, detelemref, specific). The child elements author, detelemref and specific are optional so if they are not present the content is still valid. The #PCDATA stands for “parsed character data” which is text[1].

Listing 19 shows the XML definition of the Vertex detector containing the author of this definition, the version, geometry information about the associated logical volume and two sub detectors of the LHCb detector.

Listing 19 Example of a detector element definition in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <detelem classID="2" name="Vertex" type="passive">
    <author>Author Name</author>
    <version>0.1</version>
    <geometryinfo>
      <lvname name="/dd/Geometry/lvVertex"/>
      <support name="/dd/Structure/LHCb">
        <npath value="pvVertex"/>
      </support>
    </geometryinfo>
    <detelemref classID="9999" href="vertex.xml#VertexStation01"/>
    <detelemref classID="9999" href="vertex.xml#VertexStation02"/>
  </detelem>
</DDDB>
```

The example shows that the definition of the sub detectors can be found in vertex.xml. The standard class ID for detector element is 2, but in the example the class ID of the VertexStation sub-detectors is set to 9999 which means that this sub-detector is customized by the user and contains specific information. How to customize detector elements will be described in section 8.5.3.

The geometryinfo element definition says that the Vertex detector has an associated logical volume with the given name on the transient store. The logical volume should be defined somewhere else in the XML and is retrieved automatically when the user asks for it via the IGeometryInterface returned by the geometry() accessor method. There two more tags in the geometryinfo definition. They are supporting detector element and its replica path. The supporting detector element is one of the parent detector elements sitting on the same branch in the logical detector description hierarchy. It can be a direct parent or a detector element 3 levels up. Its role is to help in finding of physical volume location in the geometry hierarchy for the detector element which is referring to it. The replica path is a sequence of numbers which is used by the algorithm looking for the physical volume location in the geometry hierarchy. This sequence is composed of the number of daughter volume IDs which must be followed from a logical volume of supporting detector elements down through the hierarchy. If it is in numeric form (rpath) then it must start counting from 0. If it is in the literal form (npath) it can be a text label but it must be unique inside the mother logical volume where it is defined and used. This label is internally converted into its proper numeric equivalent at run-time.

1. Text format of #PCDATA content in XML is verbatim so all white space characters are preserved

Finally, the `specific` child element can have anything inside its content and is provided to allow customizing of the detector elements. Users can define their own XML tags which describe very specific features or characteristics of the detector. The customizing of detector elements is described in section 8.5.3.

8.5.2.3 Defining logical volumes

To define a logical volume in XML the `logvol` element is used, see Figure 14.

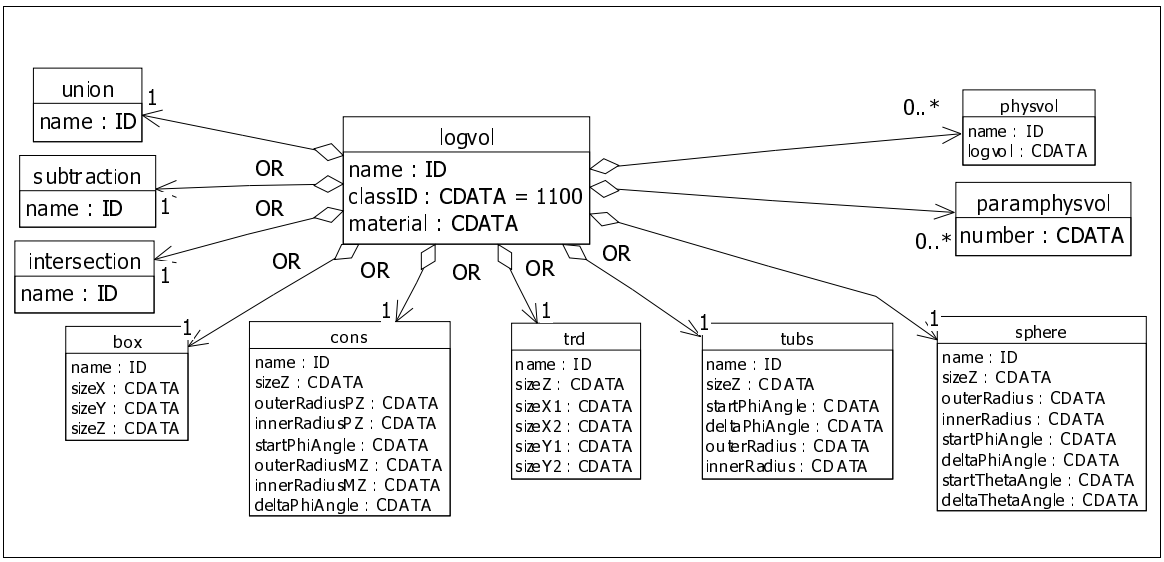


Figure 14 Content model definition of logvol element

The complete logical volume definition includes the name of the logical volume, its material, its class ID (may be omitted, because the DTD defines its fixed value), its solid (shape) and list of daughter volumes represented by physical volumes or parametric physical volumes. The content model definitions of physical and parametric physical volumes is shown in Figure 15 and Figure 16. The diagram in Figure 14 shows all attributes for solids. Figure 17 and Listing 21 show how to define solids. Listing 20 shows an example of a logical volume in XML.

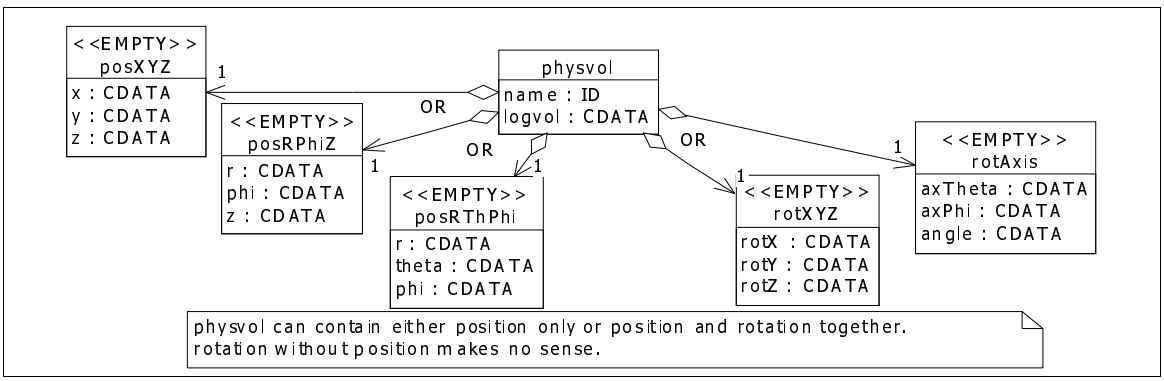


Figure 15 Content model definition for physical volume element.

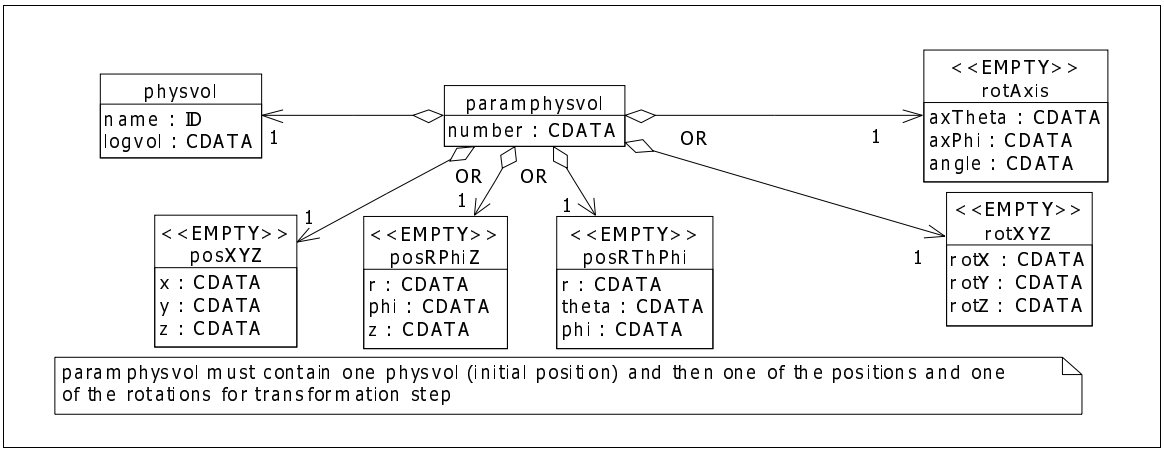


Figure 16 Content model definition for parametric physical volume element.

The example shows two logical volumes `lvLHCb` and `lvVertex`. The former volume is made of the material `Vacuum` and its solid is defined as a box with dimensions 50 x 50 x 60 meters. Then it has ascribed one daughter volume `lvVertex` positioned inside the `lvLHCb` volume without rotation and shifted along the z-axis by 200 millimetres. The `lvVertex` volume is made of material `Silicon` and its shape is defined by the solid `tubs` (cylinder) and has one parametric physical volume which places 6 daughter volumes of `lvVStation`.

Listing 20 Example of a logical volume definition in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <logvol name="lvLHCb" material="Vacuum">
    <box name="caveBox" sizeX="50*&m;" sizeY="50*&m;" sizeZ="60*&m;"/>
    <physvol name="VertexSubsystem" logvol="/dd/Geometry/lvVertex"/>
      <posXYZ x='0*&mm;' y='0*&mm;' z='200*&mm;'/>
    </physvol>
  </logvol>
  <logvol name="lvVertex" material="Silicon">
    <tubs name="vertexTubs" sizeZ="6*&m;"
          innerRadius="0*&mm;" outerRadius="20*&mm;"
          startPhiAngle="0*&degree;" deltaPhiAngle="360*&degree;"/>
    <paramphysvol number='6' >
      <physvol name="pvStation" logvol="/dd/Geometry/lvVStation"/>
        <posXYZ x='0*&mm;' y='0*&mm;' z='-200*&mm;'/>
      </physvol>
      <posXYZ x='0*&mm;' y='0*&mm;' z='50*&mm;'/>
      <rotXYZ rotX='0*&mm;' rotY='0*&mm;' rotZ='0*&mm;'/>
    </paramphysvol>
  </logvol>
</DDDB>
```

The initial position of the first daughter volume is placed 200 millimeters backwards along the Z axis relative to the center ofthe `lvVertex` mother volume. The remaining daughter

volumes are then shifted by 50 millimeters along Z axis one by one relative to the previous one without rotation.

Simple solids

Simple solids are based on the concept of **Constructive Solid Geometry** (CSG). This concept is nothing new, it has been used already by GEANT3 and GEANT4 and by the other frameworks. There are defined the solids of the basic 3D volumes such as sphere, box, cylinder, conus, toroid...

The solid definition is fundamental property of a logical volume object and thus is needed to capture this information inside XML as well. For that purpose there are defined the XML tags for the solids of the basic 3D volumes such as sphere, box, cylinder, conus, toroid... The diagram on the Figure 17 shows content model definition of the simple solids. The example of XML is shown in the Listing 21. Very important is that simple solids can't contain position and rotation definitions in the context of logical volume. When used inside logical volume the transformation makes no sense. On the other hand when used in the context of boolean solids the transformation tags are needed in order to specify the position of the solids according to each other.

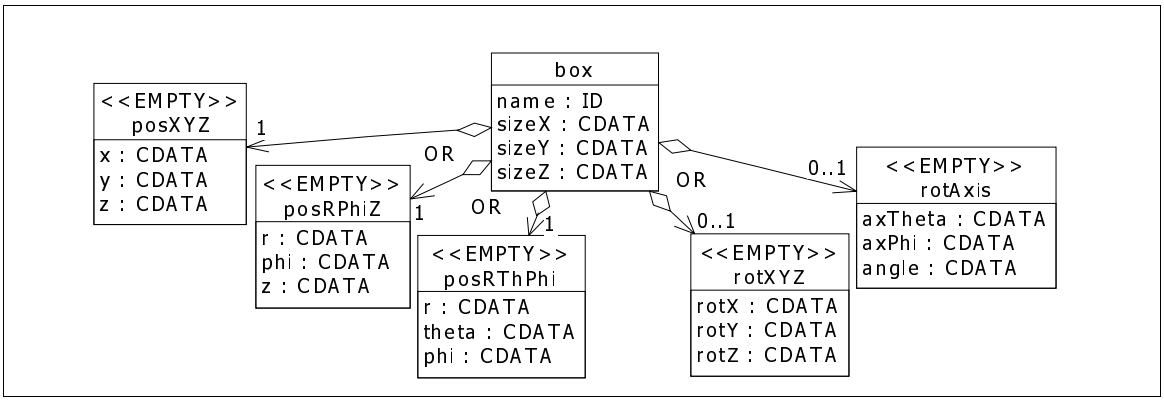


Figure 17 Content model definition of the box XML element (applies to all simple solids).

Boolean solids

The current implementation allows to create boolean solids as a union, intersection or subtraction of several simple solids. Content model is shown on the Figure 18

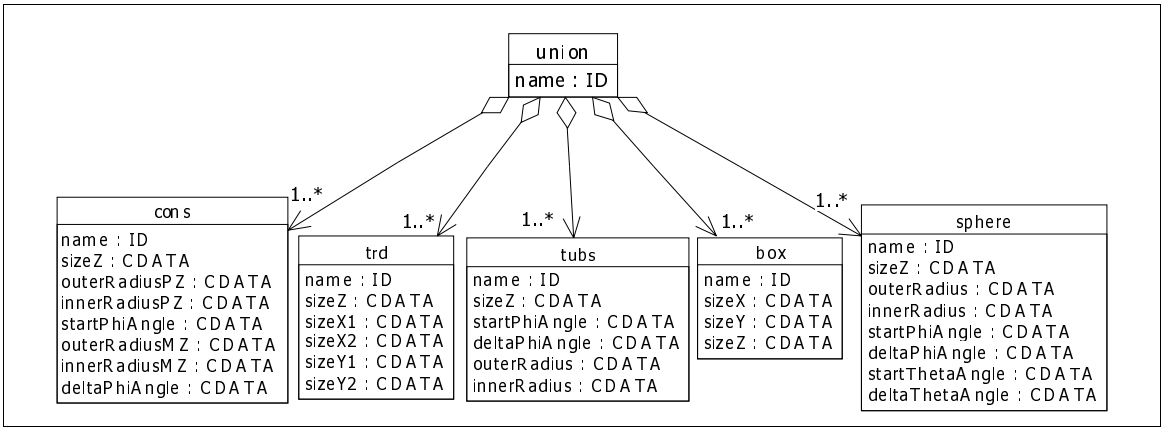


Figure 18 Content model definition of a union boolean solid (applies to all boolean solids).

The user has to start with the so called main solid and all the other solids participating in the boolean solid are positioned relative to the main solid. In this context the position and rotation attributes of solids are applied together with their dimensions attributes. Listing 21 show how to create a simple union of solids.

Listing 21 Example of boolean solid in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <logvol name="lvSampleBoolean" material="Vacuum">
    <union name="BoxUnion">
      <box name="MainBox" sizeX="10*&mm;" sizeY="10*&mm;" sizeZ="10*&mm;">
        <posXYZ x='0*&mm;' y='0*&mm;' z='-5*&mm;'/>
      </box>
      <box name="Box2" sizeX="10*&mm;" sizeY="10*&mm;" sizeZ="10*&mm;">
        <posXYZ x='0*&mm;' y='0*&mm;' z='10*&mm;'/>
      </box>
    </union>
  </logvol>
</DDDB>
```

The example shows the union of two boxes where the first one is the main solid and the second one is positioned just next to the main solid along the z-axis.

8.5.2.4 Defining material

Material is another important attribute of the logical volumes. Materials can be defined as isotopes, elements or mixtures.Elements can be optionally composed of isotopes. Mixtures

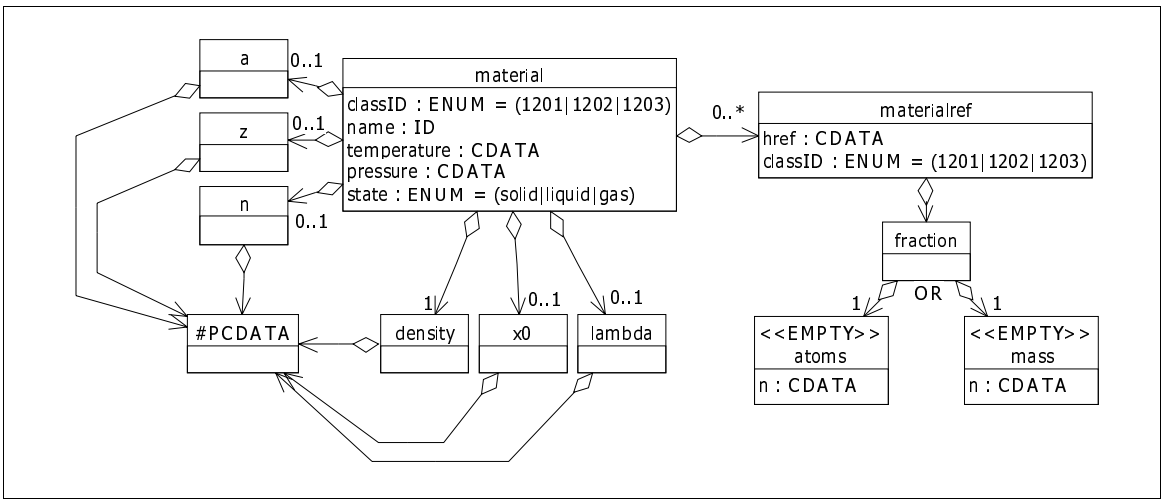


Figure 19 Content model definition of material element

can be composed of elements or of other mixtures. Composition of elements is done always by specifying the fraction of the mass for each of the isotopes of the element. For a mixture the user can specify either composition by number of atoms of by fraction of the mass for each of the elements or mixtures. The listing shows how to define a material in XML. It is not allowed to mix composition by fraction of the mass and by atoms inside the same definition of a mixture.

Listing 22 Example of a material in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <material name="Argon_CF_4_CO_2" classID="&mixture;">
    <density>2.14160E-3*&g;/&cm3;</density>
    <materialref classID="&element;" href="#Argon">
      <fraction> <mass n="0.3"/> </fraction>
    </materialref>
    <materialref classID="&mixture;" href="#CF_4">
      <fraction> <mass n="0.2"/> </fraction>
    </materialref>
    <materialref classID="&mixture;" href="#CO_2">
      <fraction> <mass n="0.5"/> </fraction>
    </materialref>
  </material>
</DDDB>
```

8.5.3 Customizing a detector element

The specific detector description can be made available to algorithms by customizing the generic detector element. Customizing is done by inheriting the specific detector class from the generic `DetectorElement`. The sub-detector specialist can provide specific answers to algorithms based on a combination of common parameters (general geometry, material etc.) and some specific parameters. For example, an algorithm may want to know what are the coordinates in the local system of reference of a given cell or wire number. The specialist can "code" the answer by using a minimal number of parameters specific to the detector structure.

Step 1: Define your detector element type

This means to provide C++ definition of your specific detector module as shown on the Listing 23 providing that the actual implementation of all the methods is done inside `MyDetector.cpp` file.

Listing 23 The C++ definition of the user defined detector element type `MyDetector`.

```
#include "Gaudi/DetectorDataSvc/DetectorElement.h"

extern const CLID& CLID_MyDetector;

class MyDetector: public DetectorElement {
public:

  double cellSize( x,y );
  void setCellSize( double size );

  double AlPlateThickness();
  void setAlPlateThickness( double thick );

  inline const CLID& clID()   { return MyDetector::classID(); }
  static const CLID& classID(){ return CLID_MyDetector; }

private:

  double m_cellSize;
  double m_AlPlateThickness;
};
```

At first the user must obtain the unique class ID for his/her own detector element type. The class ID is needed for conversion service and the corresponding converter. The methods returning the class ID must be supplied exactly as shown in the example.

The example shows further that the new detector element type will have its specific data, e.g. detector cell size. For this information we will create a user defined XML tag inside the `<specific>` section of detector element XML tag, see the next step.

Step 2: Define your detector element in XML

In this step is needed to define all the new XML tags for the detector element specific data that the user wants to access from XML data file. The definition of the new XML tags is done inside, so called, **local DTD section**. This is needed for the validation purpose. Having that definition the XML parser is then able check whether the syntax of the user defined XML tags is valid or not. The next step is to provide an information about geometry. Actually not full geometry description is done here, only the association is created telling the converter where to look for full geometrical information about this detector element. How to define geometry is shown at the next step. And finally the user has to define sub detectors or sub modules if any. Example of XML definition of `MyDetector` element type is shown in the Listing 24.

Listing 24 XML definition of `MyDetector` element type.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd" [
<!-- MyDetector cell size -->
<!ELEMENT CellSize EMPTY>
<!ATTLIST CellSize n CDATA #REQUIRED>
<!-- Thickness of the Aluminum plates -->
<!ELEMENT AluminiumPlateThickness (#PCDATA)>
]>
<DDDB>
  <detelem classID="7001" name="MyDetectorModule">
    <author>RCH</author>
    <version>0.1</version>
    <geometryinfo>
      <lvname  name = "/dd/Geometry/MyDetector/lvMyDetector" />
      <support name = "/dd/Structure/LHCb">
        <npath value = "MDM_0" />
      </support>
    </geometryinfo>
    <detelemref classID="7002" href='mysubmodule.xml#MySubModule01'/>
    <detelemref classID="7002" href='mysubmodule.xml#MySubModule02'/>
    <detelemref ="7003" href='anothermodule.xml#AnotherSubModule'/>
    <specific>
      <CellSize n="56.7*&cm;" />
      <AluminiumPlateThickness>
        4.53*&mm;
      </AluminiumPlateThickness>
    </specific>
  </detelem>
</DDDB>
```

In the piece of XML shown above are defined 2 new XML tags `AluminiumPlateThickness` and `CellSize`. They are the used inside the `<specific>` section filled with the concrete data.

The `CellSize` tag is defined as an empty tag with one attribute “n”. The tag `AluminiumPlateThickness` has no attribute but has its content. In the step 4 we will show how properly retrieve information out if these tags by methods of a user defined XML converter.

Step 3: Define the geometry of your detector element.

In the previous step the association to the geometry has defined, but the actual geometry does not exist yet. This step will fill the gap by providing all the needed information in XML.

Listing 25 XML definition of the geometry for the `MyDetector` element type.

```
<?xml version="1.0"?>
<!DOCTYPE DDDB SYSTEM "xmldb.dtd">
<DDDB>
  <catalog name="MyDetector">
    <logvolref href="#lvMyDe" />
    <logvolref href="#lvMyDeSubMod" />
    <logvolref href="#lvMyDeAnotherMod" />
  </catalog>

  <logvol name="lvMyDe" material="Vacuum">
    <box name="lvMyDeBox" sizeZ="0.8*&m;" sizeY="10*&m;" sizeX="10*&m;"/>

    <paramphysvol number="2">
      <physvol name="ppvMySM" logvol="/dd/Geometry/MyDetector/lvMyDeSubMod">
       <posXYZ x="0*&mm;" y="0*&mm;" z="-300*&mm;"/>
      </physvol>
      <posXYZ x="0*&mm;" y="0*&mm;" z="100*&mm;"/>
      <rotXYZ rotX="0*&degree;" rotY="0*&degree;" rotZ="90*&degree;"/>
    </paramphysvol>

    <physvol name="pvMyAnotherSM"
             logvol="/dd/Geometry/MyDetector/lvMyDeAnotherMod">
      <posXYZ x="0*&mm;" y="0*&mm;" z="200*&mm;" />
    </physvol>

  </logvol>
</DDDB>
```

First we define the catalog of all the logical volumes of `MyDetector`.This catalog is accessible at run-time as `/dd/Geometry/MyDetector`. Such a catalog allows modification of the XML detector description database by many people in parallel, because all the changes to the geometry structure inside this catalog can be kept local to this catalog without affecting the other subdetectors. It also makes the structure on the transient store more clear and similar to the logical structure of the detector. The catalog contains only references to logical volumes defined in the same file.

Next we have defined the logical volume for our `MyDetector` element with its shape as box and 3 daughter volumes. The first two daughter volumes are represented by parametric physical volume and the third one by normal physical volume.

In this file are missing definition for the remaining two logical volumes, but these are defined in a similar way as the one shown above.

Step 4: Write the XML converter for your detector element.

As a first step we need to create instance of the user level XML converter for `MyDetector` element type. This done by inheritance from `XmlUserDeCnv<>` templated class which is parametrized by our `MyDetector` element type, see Listing 26..

Listing 26 Making an instance of user defined XML converter for `MyDetector` element type.

```
#include "DetDesc/XmlCnvSvc/XmlUserDeCnv.h"
#include "MyDetector.h"

class XmlMyDetectorCnv : public XmlUserDeCnv<MyDetector> {
public:
  XmlMyDetectorCnv(ISvcLocator* svc);
  ~XmlMyDetectorCnv()  {}
  virtual void uStartElement( const char* const name,
                              XmlCnvAttributeList& attributes);
  virtual void uEndElement( char* const name );
  virtual void uCharacters( const char* const chars,
                            const unsigned int length);
  virtual void uIgnorableWhitespace( const char* const chars,
                                     const unsigned int length);
};

static CnvFactory<XmlMyDetectorCnv> myde_factory;
const ICnvFactory& XmlMyDetectorCnvFactory = myde_factory;

XmlMyDetectorCnv::XmlMyDetectorCnv(ISvcLocator* svc)
: XmlUserDeCnv<MyDetector>( svc, "XmlMyDetectorCnv" )
{}
```

What is important here is to define the corresponding converter factory otherwise this converter will not be invoked.

As the next step is actual implementation of the callbacks of the `IUserSax8BitDocHandler` interface. This interface is defined as shown on the Listing 27.

Listing 27 `IUserSax8BitDocHandler` interface methods

```
virtual void uStartDocument() = 0;
virtual void uEndDocument() = 0;
virtual void uCharacters( const char* const chars,
                          const unsigned int length ) = 0;
virtual void uIgnorableWhitespace( const char* const chars,
                                   const unsigned int length ) = 0;
virtual void uStartElement( const char* const name,
                            XmlCnvAttributeList& attributes) = 0;
virtual void uEndElement( const char* const name ) = 0;
```

The names of the methods are self descriptive but two of them may need more detailed explanation. The methods `uIgnorableWhitespace` and `uCharacters` are called for #PCDATA contents of an XML tag. The reason why there are two methods instead of one is that XML

specification says that white characters inside XML documents are preserved and the decision is left up to the application which should either ignore them or process them.

Listing 28 Implementation of the callback invoked at start of each XML tag inside <specific> section.

```
static std::string s_collector;

void XmlMyDetectorCnv::uStartElement(const char* const name,
                                     XmlCnvAttributeList& attributes) {
  std::string tagName( name );
  if( tagName == "CellSize" ) {
    std::string nval = attributes.getValue( "n" );
    m_dataObj->setCellSize( xmlSvc()->eval(nval) );
  } else if(tagName == “AluminiumPlateThickness” ) {
    s_collector.erase();
  } else {// Unknown tag, a warning message can be issued here}
}
```

Let’s move back to writing our XML converter. The `uStartElement` is called always when XML parser jumps onto the next XML tag inside the <specific> section. In the Listing 28 is shown the implementation of this callback.

There is a clear action there for the tag `CellSize`. We get the value of the attribute “n” by name and we evaluate its content by numerical expressions parser with checking for CLHEP units enabled and finally we set the `m_cellSize` property of our detector element.

For the tag `AluminiumPlateThickness`, however, we can’t do much at this this time because its content has not been parsed yet. What we do here is initialization of the static variable used to collect all of the characters inside the content of the `AluminiumPlateThickness` tag.

Listing 29 Implementation of callbacks needed to process content model based XML tag.

```
void XmlMyDetectorCnv::uCharacters(const char* const chars,
                                   const unsigned int length) {
  s_collector += chars;
}

void XmlMyDetectorCnv::uIgnorableWhitespace(const char* const chars,
                                            const unsigned int length) {
  s_collector += chars;
}

void XmlMyDetectorCnv::uEndElement(const char* const name) {
  std::string tagName( name );
  if(tagName == “AluminiumPlateThickness” ) {
    m_dataObj->setAluminiumPlateThickness( xmlSvc()->eval(s_collector) );
  }
}
```

The Listing 29 shows how we process #PCDATA content of our XML tag by collecting all of the characters and inside the `uEndElement` callback we finally evaluate the string expression we have collected and set the corresponding property of our detector element.

8.5.4 CLHEP units

Since Gaudi release 4, all of the packages have been made CLHEP units aware. The detector description package handles and assumes CLHEP units internally wherever they are used. The implication of this approach is that the user of the Gaudi framework must use units as well in his/her code otherwise the correct results are not guaranteed. The persistency for detector description based on XML is aware of CLHEP units as well. The use of units inside XML data files is possible due to the numerical expressions parser described section 8.5.5. This allows the user use units he/she is used to work with. May be is good to mention the list of a few units used by detector description:

Table 10 List of CLHEP units used by detector description in Gaudi

Value	Default unit
length	milimetre
angle	radian
density[ρ]	gram/centimetre^3
radiation length[X_0]	centimetre
absorption length[λ]	centimetre
A	gram/mole
weight	gram

8.5.5 Numerical expressions parser

This is a simple parser for evaluation of numerical expressions. It is available for Gaudi framework converters as well as for the user defined converters. The only difference is that in user defined converters the parser is not instantiated explicitly by the user but is accessed through `IXmlSvc` interface instead, see Listing 29 for example of its use.

The numerical expressions recognized by the parser can be composed of integers and floating point numbers assuming one of the formats:

```
100  100.  .05  0.1  1.34-e12  -23
```

Supported operations are: `+  -  *  /  unary +|-  exponent ^`

Parenthesized expressions: `1.4 * (23.4-e12 / 1.8)`

Operator precedence is: `()  unary +|-  ^  *|/  +|-`

The result is always evaluated to double value. The check for presence of CLHEP units inside the expressions is enabled by default. To suppress this behavior the call of the `eval()` method must look like:

```
xmlSvc->eval( “2*(34.5 + 1.23-e4)“, false );
```


Chapter 9
Histogram facilities

9.1 Overview

The histogram data store is one of the data stores discussed in chapter 2. Its purpose is to store statistics based data and user created objects that have a lifetime of more than a single event (e.g. histograms).

As with the other data stores, all access to data is via a service interface. In this case it is via the `IHistogramSvc` interface, which is derived from the `IDataProviderSvc` interface discussed in chapter 6. The user asks the Histogram Service to book a histogram and register it in the histogram data store. The service returns a pointer to the histogram, which can then be used to fill and manipulate the histogram

The histograms themselves are booked and manipulated using four interfaces as defined by the AIDA (Abstract Interfaces for Data Analysis) project. These interfaces are documented on the AIDA web pages: *http://wwwinfo.cern.ch/asd/lhc++/AIDA/index.html*.The GAUDI implementation uses the transient part of HTL (Histogram Template Library), provided by LHC++.

The histogram data model is shown in Figure 20. The interface `IHistogram` is a base class, which is used for management purposes. It is not a complete histogram interface, it should not be used by the users. Both interfaces `IHistogram1D` and `IHistogram2D` are derived from `IHistogram`, and use by reference the `IAxis` interface. Users can book their 1D or 2D histograms in the histogram data store in the same type of tree structure as the event data. Concrete 1D and 2D histograms derive from the `DataObject` in order to be storable

.

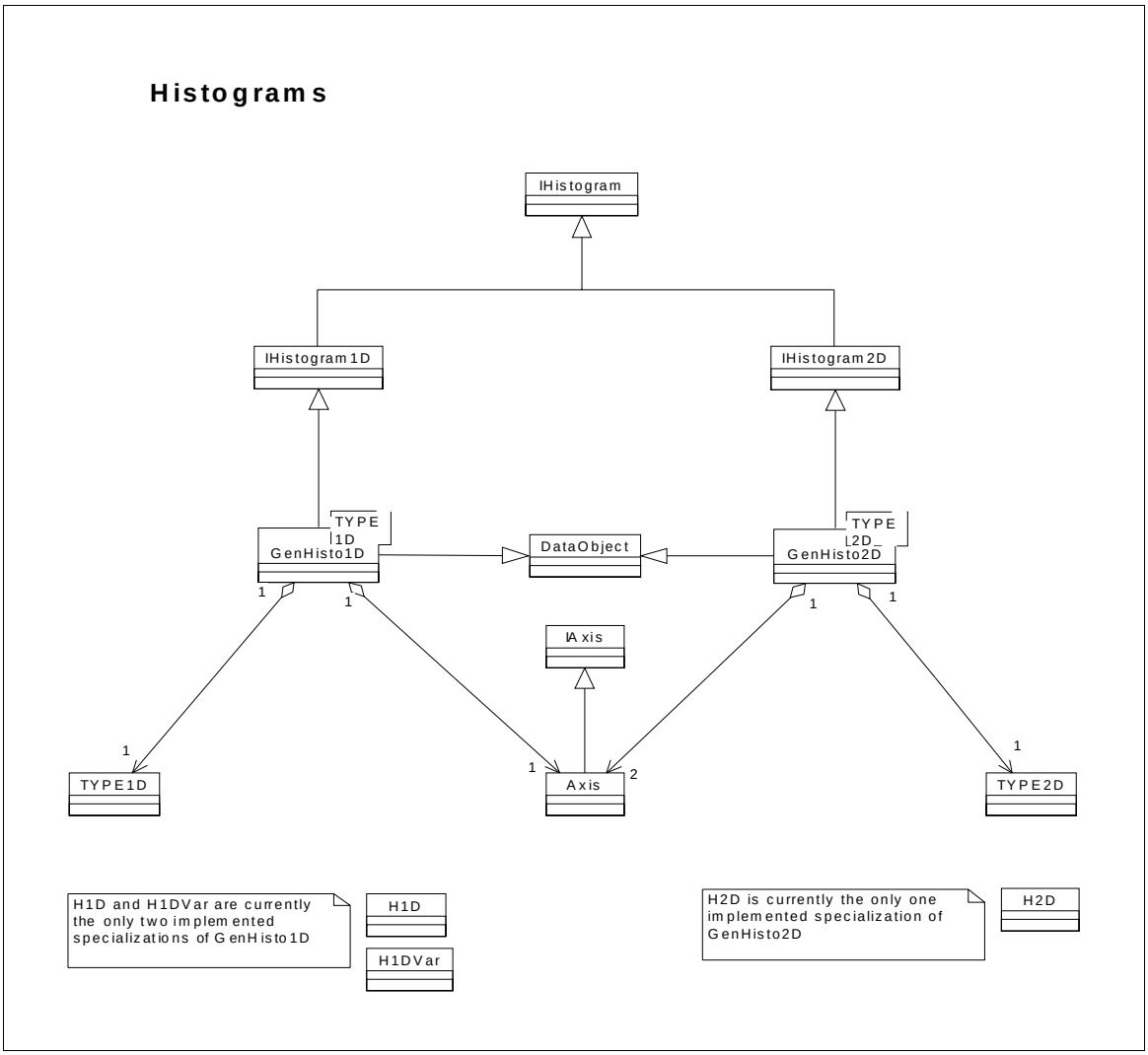


Figure 20 Histograms data model.

9.2 The Histogram service.

An instance of the histogram data service is created by the application manager. After the service has been initialised, the histogram data store will contain a root directory "`/stat`" in which users may book histograms and/or create sub-directories (for example, in the code fragment below, the histogram is stored in the subdirectory "`/stat/simple`"). A suggested naming convention for the sub-directories is given in Section 1.2.3.

As discussed in section 5.2, the `Algorithm` base class defines a member function

```
IHistogramSvc* histoSvc()
```

which returns a pointer to the `IHistogramSvc` interface of the standard histogram data service. Access to any other non-standard histogram data service (if one exists) must be sought via the `ISvcLocator` interface of the application manager as discussed in section 11.2 of Chapter 11.

9.3 Using histograms and the histogram service

An example code fragment illustrating how to book a 1D histogram and place it in a directory within the histogram data store, and a simple statement which fills that histogram is shown here:

```
// Book 1D histogram in the histogram data store
m_hTrackCount= histoSvc()->
           book( "/stat/simple", 1, “TrackCount“, 100, 0., 3000. );
SmartDataPtr<MCParticleVector> particles( eventSvc(),
                                          “/Event/MC/MCParticles” )
if ( 0 != particles ) {
  // Filling the track count histogram
  m_hTrackCount->fill(particles->size(), 1.);
}
```

The parameters of the `book` function are the directory in which to store the histogram in the data store, the histogram identifier, the histogram title, the number of bins and the lower and upper limits of the X axis. 1D histograms with fixed and variable binning are available. In the case of 2D histograms, the `book` method requires in addition the number of bins and lower and upper limits of the Y axis.

In the current implementation, the histogram identifier should be a valid HBOOK histogram identifier (number), must be unique and, in particular, must be different from any n-tuple number. These limitations are imposed by the fact that Gaudi is currently using HBOOK for histogram and n-tuple persistency.

The call to `histoSvc()->book(...)` returns a pointer to an object of type `IHistogram1D` (or `IHistogram2D` in the case of a 2D histogram). All the methods of this interface can be used to further manipulate the histogram, and in particular to fill it, as shown in the example. Note that this pointer is guaranteed to be non-null, the algorithm would have failed the initialisation step if the histogram data service could not be found. On the contrary the user variable `particles` may be null (in case of absence of Monte Carlo particles in the transient data store and in the persistent storage), and the fill statement would fail - so the value of `particles` must be checked before using it.

Algorithms which create histograms will in general keep pointers to those histograms, which they may use for filling operations. However it may be that you wish to share histograms between different algorithms. Maybe one algorithm is responsible for filling the histogram and another algorithm is responsible for fitting it at the end of the job. In this case it may be necessary to look for histograms within the store. The mechanism for doing this is identical to

the method for locating event data objects within the event data store, namely via the `IDataProviderSvc` interface, as discussed in section 6.2.

```
SmartDataPtr<IHistogram1D> hist1D( histoSvc(), "/stat/simple/1" );
if( 0 != hist1D ) {
  // Print the found histogram
  histoSvc()->print( hist1D );
}
```

9.4 Persistent storage of histograms.

An HBOOK conversion service exists which can convert objects of types `IHistogram1D` and `IHistogram2D` into a form suitable for storage in a standard HBOOK file. In order to be able to save the histograms to disk it is necessary to declare the name of the output file in the job options:

```
HistogramPersistencySvc.OutputFile  = "histo.hbook";
```

At the end of the job all the histograms in the transient histogram store will be written into this file.

On both Linux and NT, you also need to declare in the job options file the DLL containing the HBOOK conversion service:

```
ApplicationMgr.DLLs = { "HbookCnv"}
```


Chapter 10
N-tuple facilities

10.1 Overview

User data - so called n-tuples - are very similar to event data. Of course, the scope may be different: a row of an n-tuple may correspond to a track, an event or complete runs. Nevertheless, user data must be accessible by interactive tools such as PAW or Root.

Gaudi n-tuples allow to freely format structures. Later during the running phase of the program data are accumulated and written to disk.

The transient image of an n-tuple is stored in a Gaudi data store which is connected to the n-tuple service. Its purpose is to store user created objects that have a lifetime of more than a single event.

As with the other data stores, all access to data is via a service interface. In this case it is via the `INTupleSvc` interface which extends the `IDataProviderSvc` interface. In addition the interface to the n-tuple service provides methods for creating n-tuples, saving the current row of an n-tuple or retrieving n-tuples from a file. The n-tuples are derived from `DataObject` in order to be storable, and are stored in the same type of tree structure as the event data. This inheritance allows to load and locate n-tuples on the store with the same smart pointer mechanism as is available for event data items (c.f. Chapter 6).

10.2 Access to the N-tuple Service from an Algorithm.

The `Algorithm` base class defines a member function

```
INTupleSvc* ntupleSvc()
```


which returns a pointer to the `INTupleSvc` interface.

The n-tuple service provides methods for the creation and manipulation of n-tuples and the location of n-tuples within the database. The database reflects the tree structure of the n-tuple data store. Please refer to the online documentation for a description of this interface.

The top level directory of the n-tuple data store is called "/NTUPLES". The next directory layer is connected to the different output streams: e.g. "/NTuples/FILE1", where FILE1 is the logical name of the requested output file for a given stream. There can be several output streams connected to the service. In case of Persistency using HBOOK "FILE1" corresponds to the top level RZ Directory of the file (...the name given to HROPEN). From then on the tree structure is reflected with normal RZ directories.

10.3 Using the n-tuple service.

When defining an n-tuple the following steps must be performed:

- The n-tuple tags must be defined.
- The n-tuple must be booked and the tags must be declared to the n-tuple.
- The n-tuple entries have to be filled.
- The filled row of the n-tuple must be committed.
- Persistent aspects are steered by the job options.

In the following an attempt is made to explain the different steps. Please note that the n-tuple number must be unique and, in particular, that it must be different from any histogram number. This is a limitation imposed by the fact that Gaudi is currently using HBOOK for histogram and n-tuple persistency.

10.3.1 Defining n-tuple tags

When creating an n-tuple it is necessary to first define the tags to be filled in the n-tuple. Typically the tags belong to the filling algorithm and hence should be provided in the Algorithm's header file. Currently the following data types are supported: `bool`, `long`, `float` and `double`. `double` types (Fortran REAL*8) need special attention: the n-tuple structure must be defined in a way that aligns `double` types to 8 byte boundaries. In addition PAW cannot understand `double` types. The code fragment below illustrates how to define n-tuple items:

Listing 30 Definition of n-tuple tags from the `Ntuples.WriteAlg.h` example header file.

```
1:  NTuple::Item<long>              m_ntrk; // A scalar item (number)
2:  NTuple::Array<bool>             m_flag; // Vector items
3:  NTuple::Array<long>             m_index;
4:  NTuple::Array<float>            m_px, m_py, m_pz;
5:  NTuple::Matrix<long>            m_hits; // Two dimensional tag
```


10.3.2 Booking and Declaring Tags to the N-tuple

When booking the n-tuple, the previously defined tags must be declared to the the n-tuple. Before booking, the proper output stream (file) must be accessed and the target directory defined.

Listing 31 Creation of an n-tuple in a specified directory and file.

```
 1:  // Access the output file
 2:  NTupleFilePtr file1(ntupleSvc(), "/NTUPLES/FILE1");
 3:  if ( file1 )    {
 4:    if ( !ntupleSvc()->createDirectory("/NTUPLES/FILE1/MC") ) {
 5:      log << MSG::ERROR << "Cannot create directory" << endreq;
 6:    }
 7:    else {
 8:  // First: A column wise N tuple
 9:      NTuplePtr nt1(ntupleSvc(), "/NTUPLES/FILE1/MC/1");
10:      if ( !nt )    {    // Check if already booked
11:      nt=ntupleSvc()->book (col, 1, CLID_ColumnWiseTuple, "Hello World");
12:      if ( nt )    {
13:  // Add an index column
14:        status = nt->addItem ("Ntrack", m_ntrk, 0, 5000 );
15:  // Add a variable size column of type float (length=length of index col)
16:        status = nt->addItem ("px", m_ntrk, m_px);
17:        status = nt->addItem ("py", m_ntrk, m_py);
18:        status = nt->addItem ("pz", m_ntrk, m_pz);
19:  // Another one, but this time of type bool
20:        status = nt->addItem ("flg",m_ntrk, m_flag);
21:  // Another one, type integer, numerical numbers must be within [0, 5000]
22:        status = nt->addItem ("idx",m_ntrk, m_index, 0, 5000 );
23:  // Add 2-dim column: [0:m_ntrk][0:2]; numerical numbers within [0, 8]
24:        status = nt->addItem ("hit",m_ntrk, m_hits, 2, 0, 8 );
25:      }
26:      else    {   // did not manage to book the N tuple....
27:        return StatusCode::FAILURE;
28:      }
29:    }
30:  }
```

Tags which are not declared to the n-tuple are invalid and will cause an access violation at run-time. Later these tags can be used like ordinary numbers, arrays or matrices. However, there is one caveat: `double` numbers must be 8-byte aligned, otherwise HBOOK complains.

10.3.3 Filling the N-tuple

The tags should be usable just like normal data items, where

- `Items<TYPE>` are the equivalent of numbers: `bool`, `long`, `float`.
- `Array<TYPE>` are equivalent to 1 dimensional arrays: `bool[size]`, `long[size]`, `float[size]`
- `Matrix<TYPE>` are equivalent to an array of arrays or matrix: `bool[dim1][dim2]`.

There is no implicit bounds checking possible without a rather big overhead at run-time. Hence it is up to the user to ensure the arrays do not overflow.

When all entries are filled, the row must be committed, ie. the record of the n-tuple must be written.

Listing 32 Filling an n-tuple.

```
31:  m_ntrk = 0;
32:  for( MCParticleVector::iterator i = mctracks->begin(); i !=
     mctracks->end(); i++ )    {
33:    const HepLorentzVector& mom4 = (*i)->fourMomentum();
34:    m_px[m_ntrk] = mom4.px();
35:    m_py[m_ntrk] = mom4.py();
36:    m_pz[m_ntrk] = mom4.pz();
37:    m_index[m_ntrk] = cnt;
38:    m_flag[m_ntrk] = (m_ntrk%2 == 0) ? true : false;
39:    m_hits[m_ntrk][0] = 0;
40:    m_hits[m_ntrk][1] = 1;
41:    m_ntrk++;
42:  // Make sure the array(s) do not overflow.
43:    if ( m_ntrk > m_ntrk->range().distance() ) break;
44:  }
45:  // Commit N tuple row.
46:  status = ntupleSvc()->writeRecord("/NTUPLES/FILE1/MC/1");
47:  if ( !status.isSuccess() ) {
48:    log << MSG::ERROR << "Cannot fill id 1" << endreq;
49:    }
50:  }
```

10.3.4 N-tuple Persistency

A conversion service exists which can convert `NTuple` objects into a form suitable for storage in a standard HBOOK file. In order to use this facility it is necessary to add the following line in the job options file:

```
NTupleSvc.Output     = { "FILE1#<tuples.hbook>" };
//  Perstency type of the N tuple service: 6=HBOOK
NTupleSvc.Type       = 6;
```

where `<tuples.hbook>` should be replaced by the name of the file to which you wish to write the n-tuple. FILE1 one is the logical name of the output file - it could be any other string (Caveat: HBOOK only accepts directory names with less than 8 characters!).

The handling of row wise n-tuples does not differ. However, only individual items (class `NTuple::Item`) can be filled, no arrays and no matrices. Since the persistent representation of row wise n-tuples is done by floats only, the first row of each row wise n-tuple contains the type information - when looking at a row wise n-tuple with PAW make sure to start at the second event!

Chapter 11
Framework services

11.1 Overview

Services are generally sizeable components that are setup and initialized once at the beginning of the job by the framework and used by many algorithms as often as they are needed. It is not desirable in general to require more than one instance of each service. Services cannot have a “state” because there are many potential users of them so it would not be possible to guarantee that the state is preserved in between calls.

In this chapter we give describe how services are created and accessed, and then give an overview of the various services, other than the data access services, which are available for use within the Gaudi framework. The *Job Options* service, the *Message* service, the *Particle Properties* service, the *Chrono & Stat* service and the *Random Numbers* service are available in this release.

We also describe how to implement new services for use within the Gaudi environment. We look at how to code up a service, what facilities the `Service` base class provides and how a service is managed by the application manager.

11.2 Requesting and accessing services

The Application manager creates a certain number of services by default. These are the default data access services (`EventDataSvc, DetDataSvc, NTupleSvc, HistogramDataSvc`), the default data persistency services (`EventPersistencySvc, DetectorPersistencySvc, HistogramPersistencySvc`) and other framework services (`JobOptionsSvc, MessageSvc, ChronoStatSvc, RndmGenSvc, ParticlePropertiesSvc`). Additional services can be requested via the jobOptions file,

using the property `ApplicationMgr.ExtSvc`. In the example below this option is used to create a specific type of event selector and the corresponding conversion service.:

Listing 33 Job Option to create additional services

```
ApplicationMgr.ExtSvc = { “SicbEventCnvSvc”,
                        “SicbEventSelector/EventSelector”};
```

Once created, services must be accessed via their interface. The `Algorithm` base class provides a number of accessor methods for the standard framework services, listed on lines 23 to 32 of Listing 5 on page 38. Other services can be located using the `ISvcLocator` interface of the Application manager. In the example below we use the `serviceLocator()` accessor method of an algorithm to ask the `ISvcLocator` interface of the Application manager to return the `IParticlePropertySvc` interface of the Particle Properties Service:

Listing 34 Code to access the IParticlePropertySvc interface

```
#include "Gaudi/Interfaces/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
StatusCode sc = serviceLocator()->getService(
                          "ParticlePropertySvc",
                          IID_IParticlePropertySvc,
                          reinterpret_cast<IInterface*&>( m_ppSvc ));

if ( sc.isFailure) {
...
```


11.3 The Job Options Service

The Job Options Service is a mechanism which allows to configure an application at run time, without the need to recompile or relink. The options, or properties, are set via a job options file, which is read in when the Job Options Service is initialised by the Application Manager. In what follows we describe the format of the job options file, including some examples.

11.3.1 Algorithm, Tool and Service Properties

In general a concrete Algorithm, Tool or Service will have several data members which are used to control the execution of the service or algorithm. These data members can be of a basic data type (int, float, etc.) or class (Property) that encapsulates some common behaviour and higher level of functionality.

11.3.1.1 SimpleProperties

Simple properties are a set of classes that act as properties directly in their associated Algorithm, Tool or Service, replacing the corresponding basic data type instance. The primary motivation for this is to allow optional bounds checking to be applied, and to ensure that the Algorithm, Tool or Service itself doesn't violate those bounds. Available SimpleProperties are:

- int ==> IntegerProperty or SimpleProperty<int>
- double ==> DoubleProperty or SimpleProperty<double>
- bool ==> BooleanProperty or SimpleProperty<bool>)
- std::string ==> StringProperty or SimpleProperty<std::string>

and the equivalent vector classes

- std::vector<int> ==> IntegerArrayProperty or SimpleProperty< std::vector<int> >
- etc.

The use of these classes is illustrated by the `EventCounter` class.

Listing 35 EventCounter.h

```
 1: #include "Gaudi/Algorithm/Algorithm.h"
 2: #include "Gaudi/JobOptionsSvc/Property.h"
 3: class EventCounter : public Algorithm {
 4: public:
 5:     EventCounter( const std::string& name, ISvcLocator* pSvcLocator );
 6:     ~EventCounter( );
 7:     StatusCode initialize();
 8:     StatusCode execute();
 9:     StatusCode finalize();
10: private:
11:     IntegerProperty m_frequency;
12:     int m_skip;
13:     int m_total;
14: };
```


Listing 36 EventCounter.cpp

```
 1:  #include "GaudiAlg/EventCounter.h"
 2:
 3:  #include "Gaudi/MessageSvc/MsgStream.h"
 4:  #include "Gaudi/Kernel/AlgFactory.h"
 5:
 6:  static const AlgFactory<EventCounter>    Factory;
 7:  const IAlgFactory& EventCounterFactory = Factory;
 8:
 9:  EventCounter::EventCounter(const std::string& name, ISvcLocator*
10:  pSvcLocator) :
11:    Algorithm(name, pSvcLocator),
12:    m_skip ( 0 ),
13:    m_total( 0 )
14:  {
15:      declareProperty( "Frequency", m_frequency=1 );  // [1]
16:      m_frequency.setBounds( 0, 1000 );               // [2]
17:  }
18:
19:  StatusCode
20:  EventCounter::initialize()
21:  {
22:      MsgStream log(msgSvc(), name());
23:      log << MSG::INFO << name( )
24:          << ":EventCounter::initialize - Frequency: "
25:          << m_frequency << endreq;                 // [3]
26:      return StatusCode::SUCCESS;
27:  }
```

Notes:

1. A default value may be specified when the property is declared.
2. Optional upper and lower bounds may be set (see later).
3. The value of the property is accessible directly using the property itself.

In the Algorithm constructor, when calling declareProperty, you can optionally set the bounds using any of:

```
setBounds( const T& lower, const T& upper );
setLower ( const T& lower );
setUpper ( const T& upper );
```

There are similar selectors and modifiers to determine whether a bound has been set etc., or to clear a bound.

```
bool hasLower( )
bool hasUpper( )
T lower( )
T upper( )
void clearBounds( )
void clearLower( )
void clearUpper( )
```


You can set the value using the "=" operator or the set functions

```
bool set( const T& value )
bool setValue( const T& value )
```

The function value indicates whether the new value was within any bounds and was therefore successfully updated. In order to access the value of the property, use:

```
m_property.value( );
```

In addition there's a cast operator, so you can also use m_property directly instead of m_property.value().

11.3.1.2 CommandProperty

`CommandProperty` is a subclass of `StringProperty` that has a handler that is called whenever the value of the property is changed. Currently that can happen only during the job initialization so it is not terribly useful. Alternatively, an Algorithm could set the property of one of its sub-algorithms. However, it is envisaged that GAUDI will be extended with a scripting language such that properties can be modified during the course of execution.

The relevant portion of the interface to CommandProperty is:

```
class CommandProperty : public StringProperty {
public:
  [...]
  virtual void handler( const std::string& value ) = 0;
  [...]
};
```

Thus subclasses should override the `handler()` member function, which will be called whenever the property value changes. A future development is expected to be a ParsableProperty (or something similar) that would offer support for parsing the string.

11.3.2 Job options file format

The job options file has a well-defined syntax (similar to a simplified C++-Syntax) without datatypes. The data types are recognised by the “Job Options Compiler”, which interprets the job options file according to the syntax (described in Appendix C together with possible compiler error codes).

The job options file is an ASCII-File, composed logically of a series of statements. The end of a statement is signaled by a semicolon “;“ - as in C++.

Comments are the same as in C++, with ’/ /’ until the end of the line, or between ’/*’ and ’*/’.

There are four constructs which can be used in a job options file:

- Assignment statement
- Append statement

- Include directive
- Platform dependent excution directive

Assignment statement

An assignment statement assigns a certain value (or a vector of values) to a property of an object or identifier. An assignment statement has the following structure:

```
<Object / Identifier> . < Propertyname > = < value >;
```

The first token (`Object / Identifier`) specifies the name of the object whose property is to be set. This must be followed by a dot ('.')

The next token (`Propertyname`) is the name of the option to be set, as declared in the `declareProperty()` method of the `IProperty` interface. This must be followed by an assign symbol ('=').

The final token (`value`) is the value to be assigned to the property. It can be a vector of values, in which case the values are enclosed in array brackets ('{','}'), and separated by commas (,). The token must be terminated by a semicolon (';').

The type of the value(s) must match that of the variable whose value is to be set, as declared in `declareProperty()`. The following types are recognised:

Boolean-type, written as true or false.

e.g. `true; false;`

Integer-type, written as an integer value (containing one or more of the digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9')

e.g.: `123; -923;` or in scientific notation, e.g.: `12e2;`

Real-type (similar to `double` in C++), written as a real value (containing one or more of the digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' followed by a dot '.' and optionally one or more of digits again)

e.g.: `123.; -123.45;` or in scientific notation, e.g. `12.5e7;`

String type, written within a pair of double quotes (' " ')

e.g.: `"I am a string";` (Note: strings without double quotes are not allowed!)

Vector of the types above, within array-brackets ('{', '}'), separated by a comma (',')

e.g.: `{true, false, true};`
e.g.: `{124, -124, 135e2};`
e.g.: `{123.53, -23.53, 123., 12.5e2};`
e.g.: `{"String 1", "String 2", "String 3"};`

A single element which should be stored in a vector must be within array-brackets without a comma

e.g. `{true};`
e.g. `{"String"};`

Please note that it is not necessary any more to define in the job options file the type for the value which is used. The job options compiler will recognize by itself the type of the value and will handle this type correctly.

Append Statement

Because of the possibility of including other job option files (see below), it is sometimes necessary to extend a vector of values already defined in the other job option file. This functionality is provided be the append statement.

An append statement has the following syntax:

```
<Object / Identifier> . < Propertyname > += < value >;
```

The only difference from the assignment statement is that the append statement requires the '+=' symbol instead of the '=' symbol to separate the `Propertyname` and `value` tokens.

The value must be an array of one or more values

e.g. `{true};`
e.g. `{"String"};`
e.g.: `{true, false, true};`
e.g.: `{124, -124, 135e2};`
e.g.: `{123.53, -23.53, 123., 12.5e2};`
e.g.: `{"String 1", "String 2", "String 3"};`

The job options compiler itself tests if the object or identifier already exists (i.e. has already been defined in an included file) and the type of the existing property. If the type is compatible and the object exists the compiler appends the value to the existing property.

Including other Job Option Files

It is possible to include other job option files in order to use pre-defined options for certain objects. This is done using the `#include` directive:

```
#include "filename.ext"
```

The "`filename.ext`" can also contain the path where this file is located. The include directive can be placed anywhere in the job option file, but it is strongly recommended to place it at the very top of the file (as in C++).

It is possible to use environment variables in the `#include` statement, either standalone or as part of a string. Both Unix style ("`$environmentvariable`") and Windows style ("`%environmentvariable%`") are understood (on both platforms!)

As mentioned above, you can append values to vectors defined in an included job option file. The interpreter creates these vectors at the moment he interprets the included file, so you can only append elements defined in a file included before the append-statement!

As in C/C++, an included job option file can include other job option files. The compiler checks itself whether the include file is already included or not, so there is no need for `#ifndef` statements as in C or C++ to check for multiple including.

Sometimes it is necessary to over-ride a value defined previously (maybe in an include file). This is done by using an assign statement with the same object and propertyname. The last value assigned is the valid value!

11.3.2.1 Platform dependent execution

The possibility exists to execute statements only according to the used platform. Statements within platform dependent clauses are only executed if they are asserted to the current used platform.:

```
#ifdef WIN32
(Platform-Dependent Statement)
#else (optional)
(Platform-Dependent Statement)
#endif
```

Only the variable `WIN32` is defined! An `#ifdef WIN32` will check if the used platform is a Windows platform. If so, it will execute the statements until an `#endif` or an optional `#else`. On non-Windows platforms it will execute the code within `#else` and `#endif`. Alternatively one directly can check for a non-Windows platform by using the `#ifndef WIN32` clause.

Examples

We have already seen an example of a job options file in Listing 2 in Chapter 4.

11.4 The Standard Message Service

One of the components directly visible to an algorithm object is the message service. The purpose of this service is to provide facilities for the logging of information, warnings, errors etc. The advantage of introducing such a component, as opposed to using the standard `std::cout` and `std::cerr` streams available in C++ is that we have more control over what is printed and where it is printed. These considerations are particularly important in an online environment.

The Message Service is configurable via the job options file to only output messages if their "activation level" is equal to or above a given "output level". The output level can be configured with a global default for the whole application:

```
// Set output level threshold (2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL)
MessageSvc.OutputLevel = 4;
```

and / or locally for a given client object (e.g. myAlgorithm):

```
myAlgorithm.OutputLevel = 2;
```

Any object wishing to print some output should (must) use the message service. A pointer to the `IMessageSvc` interface of the message service is available to an algorithm via the accessor method `msgSvc()`, see section 5.2. It is of course possible to use this interface directly, but a utility class called `MsgStream` is provided which should be used instead.

11.4.1 The MsgStream utility

The `MsgStream` class is responsible for constructing a `Message` object which it then passes onto the message service. Where the message is ultimately sent to is decided by the message service.

In order to avoid formatting messages which will not be sent because the verboseness level is too high, a `MsgStream` object first checks to see that a message will be printed before actually constructing it. However the threshold for a `MsgStream` object is not dynamic, i.e. it is set at creation time and remains the same. Thus in order to keep synchronized with the message service, which in principle could change its printout level at any time, `MsgStream` objects should be made locally on the stack when needed. For example, if you look at the listing of the `HistoAlgorithm` class (see also Listing 37 below) you will note that `MsgStream` objects are instantiated locally (i.e. not using `new`) in all three of the `IAlgorithm` methods and thus are destructed when the methods return. If this is not done messages may be lost, or too many messages may be printed.

The `MsgStream` class has been designed to resemble closely a normal stream class such as `std::cout`, and in fact internally uses an `ostrstream` object. All of the `MsgStream` member functions write unformatted data; formatted output is handled by the insertion operators.

An example use of the MsgStream class is shown below.

Listing 37 Use of a MsgStream object.

```
 1:  #include “Gaudi/MessageSvc/MgsStream.h”
 2:
 3:  StatusCode myAlgo::finalize() {
 4:    StatusCode status = Algorithm::finalise();
 5:    MsgStream log(msgSvc(), name());
 6:    if ( status.isFailure() ) {
 7:      // Print a two line message in case of failure.
 8:      log << MSG::ERROR << “ Finalize failed” << endl
 9:          << “Error initializing Base class.” << endreq;
10:    }
11:    else {
12:      log << MSG::DEBUG << “Finalize completed successfully” << endreq;
13:    }
14:    return status;
15:  }
```

When using the MsgStream class just think of it as a configurable output stream whose activation is actually controlled by the first word (message level) and which actually prints only when “endreq” is supplied. For all other functionality simply refer to the C++ ostream class.

The “activation level” of the MsgStream object is controlled by the first expression, e.g. MSG::ERROR or MSG::DEBUG in the example above. Possible values are given by the enumeration below:

```
enum MSG::Level { VERBOUS, DEBUG, INFO, WARNING, ERROR, FATAL };
```

Thus the code in Listing 37 will produce NO output if the print level of the message service is set higher than MSG::ERROR. In addition if the service’s print level is lower than or equal to MSG::DEBUG the “Finalize completed successfully” message will be printed (assuming of course it was successful).

User interface

What follows is a technical description of the part of the MsgStream user interface most often seen by application developers. Please refer to the header file for the complete interface.

Insertion Operator

The MsgStream class overloads the ’<<‘ operator as described below.

`MsgStream& operator <<(TYPE arg);`

Insertion operator for various types. The argument is only formatted by the stream object if the print level is sufficiently high and the stream is active. Otherwise the insertion operators simply return. Through this mechanism extensive debug printout does not cause large run-time overheads. All common base types such as char, unsigned char, int, float, etc. are supported

`MsgStream& operator <<(MSG::Level level);`

This insertion operator does not format any output, but rather (de)activates the stream's formatting and forwarding engine depending on the value of `level`.

Accepted Stream Manipulators

The `MsgStream` specific manipulators are presented below, e.g. `endreq`: `MsgStream& endreq(MsgStream& stream)`. Besides these, the common `ostream` and ios manipulators such as `std::ends`, `std::endl`,... are also accepted.

endl Inserts a newline sequence. Opposite to the ostream behaviour this manipulator does not flush the buffer. Full name: `MsgStream& endl(MsgStream& s)`

ends Inserts a null character to terminate a string. Full name: `MsgStream& ends(MsgStream& s)`

flush Flushes the stream's buffer but does not produce any output! Full name: `MsgStream& flush(MsgStream& s)`

endreq Terminates the current message formatting and forwards the message to the message service. If no message service is assigned the output is sent to `std::cout`. Full name: `MsgStream& endreq(MsgStream& s)`

11.5 The Particle Properties Service

The Particle Property service is a utility to find information about a named particle's Geant3 ID, Jetset / Pythia ID, Geant3 tracking type, charge, mass or lifetime. The database used by the service can be changed, but by default is the same as that used by SICb. Note that the units conform to the CLHEP convention, in particular MeV for masses and ns for lifetimes. Any comment to the contrary in the code is just a leftover which has been overlooked!

11.5.1 Initialising and Accessing the Service

This service is created by default by the application manager, with the name "ParticlePropertySvc". Listing 34 on page 98 shows how to access this service from within an algorithm.

11.5.2 Service Properties

The Particle Property Service currently only has one property: `ParticlePropertiesFile`. This string property is the name of the database file that should be used by the service to build up its list of particle properties. The default value of this property, on all platforms, is `$LHCBDBASE/cdf/particle.cdf`

11.5.3 Service Interface

The service implements the `IParticlePropertySvc` interface which must be included when a client wishes to use the service - the file to include is `Gaudi/Interfaces/IParticlePropertySvc.h`

The service itself consists of one STL vector to access all of the existing particle properties, and three STL maps, one to map particles by name, one to map particles by Geant3 ID and one to map particles by stdHep ID.

Although there are three maps, there is only one copy of each particle property and thus each property must have a unique particle name and a unique Geant3 ID. The third map does not contain all particles contained in the other two maps; this is because there are particles known to Geant but not to stdHep, such as Deuteron or Cerenkov. Although retrieving particles by name should be sufficient, the second and third maps are there because most of the data generated by SICb stores a particle's Geant3 ID or stdHep ID, and not the particle's name. These maps speed up searches using the IDs.

The IParticlePropertySvc interface provides the following functions:

Listing 38 The IParticlePropertySvc interface.

```
// IParticlePropertySvc interface:
// Create a new particle property.
// Input: particle, String name of the particle.
// Input: geantId, Geant ID of the particle.
// Input: jetsetId, Jetset ID of the particle.
// Input: type, Particle type.
// Input: charge, Particle charge (/e).
// Input: mass, Particle mass (MeV).
// Input: tlife, Particle lifetime (ns).
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( const std::string& particle, int geantId, int
jetsetId, int type, double charge, double mass, double tlife );

// Create a new particle property.
// Input: pp, a particle property class.
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( ParticleProperty* pp );

// Get a const reference to the begining of the map.
virtual const_iterator begin() const;

// Get a const reference to the end of the map.
virtual const_iterator end() const;

// Get the number of properties in the map.
virtual int size() const;

// Retrieve a property by geant id.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( int geantId );

// Retrieve a property by particle name.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( const std::string& name );

// Retrieve a property by StdHep id
// Pointer is 0 if no property found.
virtual ParticleProperty* findByStdHepID( int stdHepId );

// Erase a property by geant id.
virtual StatusCode erase( int geantId );

// Erase a property by particle name.
virtual StatusCode erase( const std::string& name );

// Erase a property by StdHep id
virtual StatusCode eraseByStdHepID( int stdHepId );
```


The IParticlePropertySvc interface also provides some typedefs for easier coding:

```
typedef ParticleProperty* mapped_type;
typedef std::map< int, mapped_type, std::less<int> > MapID;
typedef std::map< std::string, mapped_type, std::less<std::string> > MapName;
typedef std::map< int, mapped_type, std::less<int> > MapStdHepID;
typedef IParticlePropertySvc::VectPP VectPP;
typedef IParticlePropertySvc::const_iterator const_iterator;
typedef IParticlePropertySvc::iterator iterator;
```

11.5.4 Examples

Below are some extracts of code from the example in `GaudiExamples/ParticleProperties`, to show how one might use the service:

Listing 39 Code fragment to find particle properties by particle name.

```
// Try finding particles by the different methods
 log << MSG::INFO << "Trying to find properties by Geant3 ID..." << endreq;
 ParticleProperty* pp1 = m_ppSvc->find( 1 );
 if ( pp1 ) log << MSG::INFO << *pp1 << endreq;
 log << MSG::INFO << "Trying to find properties by name..." << endreq;
 ParticleProperty* pp2 = m_ppSvc->find( "e+" );
 if ( pp2 ) log << MSG::INFO << *pp2 << endreq;
 log << MSG::INFO << "Trying to find properties by StdHep ID..." << endreq;
 ParticleProperty* pp3 = m_ppSvc->findByStdHepID( 521 );
 if ( pp3 ) log << MSG::INFO << *pp3 << endreq;
```

Listing 40 Code fragment showing how to use the map iterators to access particle properties.

```
// List all properties
log << MSG::DEBUG << "Listing all properties..." << endreq;
for( IParticlePropertySvc::const_iterator i = m_ppSvc->begin();
       i != m_ppSvc->end(); i++ ) {
  if ( *i ) log << *(*i) << endreq;
}
```


11.6 The Chrono & Stat service

The Chrono & Stat service provides a facility to do time profiling of code (*Chrono* part) and to do some statistical monitoring of simple quantities (*Stat* part). The service is created by default by the Application Manager, with the name "`ChronoStatSvc`" and service ID `extern const CLID& IID_IChronoStatSvc` To access the service from inside an algorithm, the member function `chronoSvc()`is provided. The job Options to configure this service are described in Appendix B, Table B.15.

11.6.1 Code profiling

Profiling is performed by using the `chronoStart()` and `chronoStop()` methods inside the codes to be profiled, e.g:

```
/// ...
IChronoStatSvc* svc = ...
///  start
svc->startChrono( "Some Tag" );
/// here some user code are placed:
...
/// stop
svc->stopChrono( "SomeTag" );
```

The profiling information accumulates under the tag name given as argument to these methods. The service measures the time elapsed between subsequent calls of `startChrono()` and `endChrono()` with the same tag. The latter is important, since in the sequence of calls below, only the elapsed time between lines 3 and 5 lines and between lines 7 and 9 lines would be accumulated.:

```
 1:  svc->endChrono("Tag");
 2:  svc->endChrono("Tag");
 3:  svc->startChrono("Tag");
 4:  svc->startChrono("Tag");
 5:  svc->endChrono("Tag");
 6:  svc->endChrono("Tag");
 7:  svc->startChrono("Tag");
 8:  svc->startChrono("Tag");
 9:  svc->endChrono("Tag");
```

The profiling information could be printed either directly using the `printChrono()` method of the service, or in the summary table of profiling information at the end of the job.

11.6.2 Statistical monitoring

Statistical monitoring is performed by using the `stat()` method inside user code:

```
1: /// ... Flag and Weight to be accumulated:
2: svc->stat( " Number of Tracks " , Flag , Weight );
```

The statistical information contains the "accumulated" *flag*, which is the sum of all *Flag*s for the given tag, and the "accumulated" *weight*, which is the product of all *Weight*s for the given tag. The information is printed in the final table of statistics.

In some sense the profiling could be considered as statistical monitoring, where the variable *Flag* equals the elapsed time of the process.

11.6.3 `Chrono` and `Stat` helper classes

To simplify the usage of the Chrono & Stat Service, two helper classes were developed: `class Chrono` and `class Stat`. Using these utilities, one hides the communications with Chrono & Stat Service and provides a more friendly environment.

11.6.3.1 Chrono

`Chrono` is a small helper class which invokes the `startChrono()` method in the constructor and the `endChrono()` method in the destructor. It must be used as an *automatic local object*.

It performs the profiling of the code between its own creation and the end of the current scope, e.g:

```
1: #include Gaudi/ChronoStatSvc/Chrono.h
2: /// ...
3: { // begin of the scope
4:    Chrono chrono( svc , "ChronoTag" ) ;
5:    /// some codes:
6:     ...
7:    ///
8: } // end of the scope
9: /// ...
```

If the Chrono & Stat Service is not accessible, the *Chrono* object does nothing

11.6.3.2 Stat

`Stat` is a small helper class, which invokes the `stat()` method in the constructor.

```
 1: Gaudi/ChronoStatSvc/Stat.h
 2: /// ...
 3: Stat stat( svc , "StatTag" , Flag , Weight ) ;
 4: /// ...
```

If the Chrono Stat Service is not accesible, the `Stat` object does nothing.

11.6.4 Performance considerations

The implementation of the Chrono & Stat Service uses two `std::map` containers and could generate a performance penalty for very frequent calls. Usually the penalty is small relative to the elapsed time of algorithms, but it is worth avoiding both the direct usage of the Chrono & Stat Service as well as the usage of it through the `Chrono` or `Stat` utilities inside internal loops:

```
 1: /// ...
 2: {  /// begin of the scope
 3: Chrono chrono( svc , "Good Chrono");  /// OK
 4: long double a = 0 ;
 5: for(  long i = 0 ; i < 1000000 ; ++i )
 6:  {
 7:  Chrono chrono( svc , "Bad Chrono");   /// not OK
 8:  /// some codes :
 9:    a += sin( cos( sin( cos( (long double) i ) ) ) );
10:  ///  end of codes
11: Stat   stat  ( svc , "Bad Stat", a ); /// not OK
12:  }
13: Stat   stat  ( svc , "Good Stat", a); /// OK
14: } ///  end of the scope!
15: /// ...
```


11.7 The Random Numbers Service

When generating random numbers two issues must be considered:

- reproducibility and
- randomness of the generated numbers.

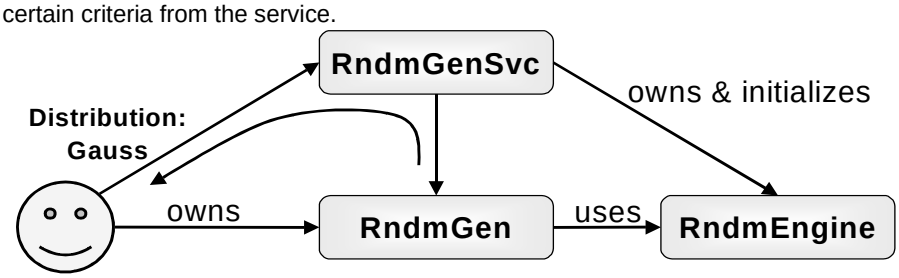
In order to ensure both, GAUDI implements a single service ensuring that these criteria are met. The encapsulation of the actual random generator into a service has several advantages:

- Random seeds are set by the framework. When debugging the detector simulation, the program could start at any event independent of the events simulated before. Unlike the random number generators that were known from CERNLIB, the state of modern generators is no longer defined by one or two numbers, but rather by a fairly large set of numbers. To ensure reproducibility the random number generator must be initialized for every event.
- The distribution of the random numbers generated is independent of the random number engine behind. Any distribution can be generated starting from a flat distribution.
- The actual number generator can easily be replaced if at some time in the future better generators become available, without affecting any user code.

The implementation of both generators and random number engines are taken from CLHEP. The default random number engine used by GAUDI is the RanLux engine of CLHEP with a luxury level of 3. This ensures that packages used to simulate the detector, such as GEANT4, use the same mechanism to generate random numbers. Figure 21 shows the general architecture of the GAUDI random number service. The client interacts with the service in the following way:

- The client requests a generator from the service, which is able to produce a generator according to a requested distribution. The client then retrieves the requested generator.
- Behind the scenes, the generator service creates the requested generator and initializes the object according to the parameters. The service also supplies the shared random number engine to the generator.
- After the client has finished using the generator, the object must be released in order to inhibit resource leaks.

Figure 21 The architecture of the random number service. The client requests a random number generator satisfying certain criteria from the service.



There are many different distributions available. The shape of the distribution must be supplied as a parameter when the generator is requested by the user.

Currently implemented distributions include the following:
(see also the header file `Gaudi/RndmGenSvc/RandomGenerators.h` for a description of the parameters to be supplied)

- Generate random bit patterns with parameters `Rndm::Bit()`
- Generate a flat distribution with boundaries [`min, max`] with parameters:
  `Rndm::Flat(double min, double max)`
- Generate a gaussian distribution with parameters: `Rndm::Gauss(double mean, double sigma)`
- Generate a poissonian distribution with parameters: `Rndm::Poisson(double mean)`
- Generate a binomial distribution according to `n` tests with a probability `p` with parameters: `Rndm::Binomial(long n, double p)`
- Generate an exponential distribution with parameters:
  `Rndm::Exponential(double mean)`
- Generate a Chi**2 distribution with `n_dof` degrees of freedom with parameters:
  `Rndm::Chi2(long  n_dof)`
- Generate a Breit-Wigner distribution with parameters:
  `Rndm::BreitWigner(double mean, double gamma)`
- Generate a Breit-Wigner distribution with a cut-off with parameters:
  `Rndm::BreitWignerCutOff (mean, gamma, cut-off)`
- Generate a Landau distribution with parameters:
  `Rndm::Landau(double mean, double sigma)`
- Generate a user defined distribution. The probability density function is given by a set of descrete points passed as a vector of doubles:
  `Rndm::DefinedPdf(const std::vector<double>& pdf, long intpol)`

Clearly the supplied list of possible parameters is not exhaustive, but probably represents most needs. The list only represents the present content of generators available in CLHEP and can be updated in case other distributions will be implemented.

Since there is a danger that the interfaces are not released, a wrapper is provided that automatically releases all resources once the object goes out of scope. This wrapper allows the use of the random number service in a simple way. Typically there are two different usages of this wrapper:

- Within the user code only a series of numbers is required once, ie. not every event. In this case the object is used locally and resources are released immediately after use. This example is shown in Listing 41.

- One or several random numbers are required for the processing of every event. En example is shown in Listing 42 and Listing 43

Listing 41 Example of the use of the random number generator for filling a histogram with a gaussian distribution within a standard GAUDI algorithm.

```
1:   Rndm::Numbers gauss(randSvc(), Rndm::Gauss(0.5,0.2));
2:   if ( gauss )    {
3:     IHistogram1D* his = histoSvc()->book("/stat/2","Gaussian",40,0.,3.);
4:     for ( long i = 0; i < 5000; i++ )
5:       his->fill(gauss(), 1.0);
6:   }
```

Listing 42 Example of the use of the random number generator for filling a histogram with a gaussian distribution within a standard GAUDI algorithm. The wrapper to the generator is declared within the header file of the algorithm, hence it is part of the Algorithm itself

```
1:  #include ìGaudi/RandmGenSvc/RndmGenerators.hî
1:  class myAlgorithm : public Algorithm  {
1:    Rndm::Numbers m_gaussDist;
2:  ...
3:  };
```

Listing 43 When part of the algorithm, the generator must be initialized before being used. Afterwards the usage is identical to the example described in .Listing 41.

```
1:   StatusCode sc=m_gaussDist.initialize(randSvc(), Rndm::Gauss(0.5,0.2));
1:   if ( !status.isSuccess() )  {
2:     // put error handling code here...
3:   }
```

There are a few points to be mentioned in order to ensure the reproducability:

- Do not keep numbers across events. If you need a random number ask for it. Usually caching does more harm than good. If there is a performance penalty, it is better to find a more generic solution.
- Do not access the RndmEngine directly.
- Do not manipulate the engine. The random seeds should only be set by the framework on an event by event basis.

11.8 Developing new services

11.8.1 The Service base class

Within Gaudi we use the term "Service" to refer to a class whose job is to provide a set of facilities or utilities to be used by other components. In fact we mean more than this because a concrete service must derive from the `Service` base class and thus has a certain amount of predefined behaviour; for example it has `intialize()` and `finalize()` methods which are invoked by the application manager at well defined times.

Figure 22 shows the inheritance structure for an example service called `SpecificService` (!). The key idea is that a service should derive from the `Service` base class and additionally implement one or more pure abstract classes (interfaces) such as `IConcreteSvcType1` and `IConcreteSvcType2` in the figure.

As discussed above, it is necessary to derive from the Service base class so that the concrete service may be made accessible to other Gaudi components. The actual facilities provided by the service are available via the interfaces that it provides. For example the `ParticleProperties` service implements an interface which provides methods for retrieving, for example, the mass of a given particle. In figure 22 the service implements two interfaces each of two methods.

A component which wishes to make use of a service makes a request to the application manager. Services are requested by a combination of name, and interface type, i.e. an algorithm would request specifically either `IConcreteSvcType1` or `IConcreteSvcType2`.

The identification of what interface types are implemented by a particular class is done via the `queryInterface` method of the `IInterface` interface. This method must be implemented in the concrete service class. In addition the `initialize()` and `finalize()` methods should be implemented. After initialization the service should be in a state where it may be used by other components.

The service base class offers a number of facilities itself which may be used by derived concrete service classes:

- Properties are provided for services just as for algorithms. Thus concrete services may be fine tuned by setting options in the job options file.
- A `serviceLocator` method is provided which allows a component to request the use of other services which it may need.
- A message service.

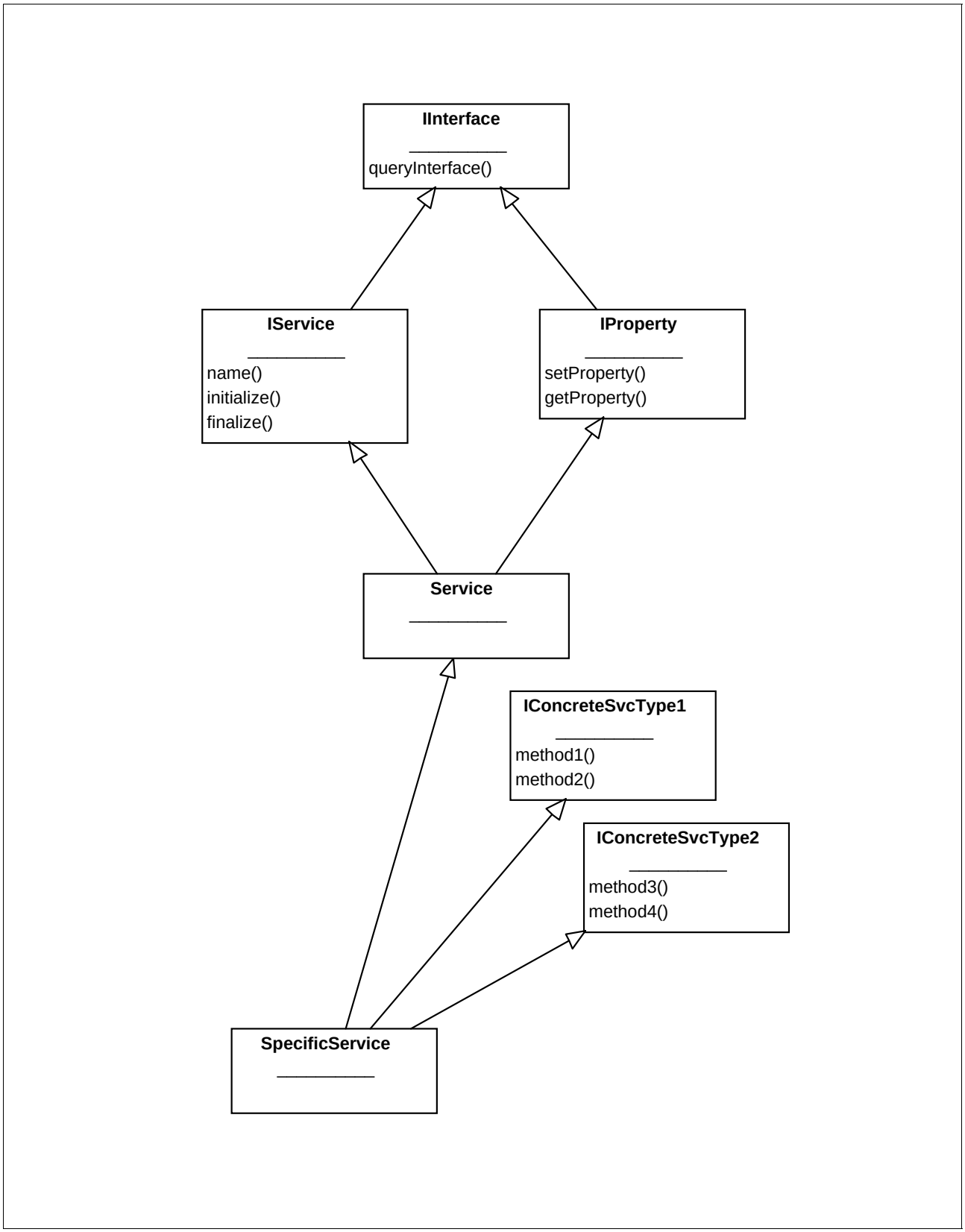


Figure 22 Implementation of a concrete service class. Though not shown in the figure, both of the IConcreteSvcType interfaces are derived from IInterface.

11.8.2 Implementation details

The following is essentially a checklist of the minimal code required for a service.

1. Define the interfaces:

Listing 44 An interface class

```
#include "Gaudi/interfaces/IInterface.h"

class IConcreteSvcType1 : virtual public IInterface {
public:
  void method1() = 0;
  int method2() = 0;
}


#include "IConcreteSvcType1.h"

const IID& IID_IConcreteSvcType1 = 143; // UNIQUE within LHCb !!
```

2. Derive the concrete service class from the `Service` base class.
3. Implement the `queryInterface()` method.
4. Implement the `initialize()` method. Within this method you should make a call to `Service::initialize()` as the first statement in the method and also make an explicit call to `setProperties()` in order to read the service's properties from the job options (note that this is different from Algorithms, where the call to `setProperties()` is done in the base class).

Listing 45 A minimal service implementation

```
#include "Gaudi/Kernel/Service.h"
#include "IConcreteSvcType1.h"
#include "IConcreteSvcType2.h"

class SpecificService : public Service,
                        virtual public IConcreteSvcType1,
                        virtual public IConcreteSvcType2 {
public:
  // Constructor of this form required:
  SpecificService(const std::string& name, ISvcLocator* sl);

  queryInterface(constIID& riid, void** ppvIF);
};


// Factory for instantiation of service objects
static SvcFactory<SpecificService> s_factory;
const ISvcFactory& SpecificServiceFactory = s_factory;

// UNIQUE Interface identifiers defined elsewhere
extern const IID& IID_IConcreteSvcType1;
extern const IID& IID_IConcreteSvcType2;

// queryInterface
StatusCode SpecificService::queryInterface(const IID& riid, void** ppvIF) {
  if(IID_IConcreteSvcType1 == riid) {
    *ppvIF = dynamic_cast<IConcreteSvcType1*> (this);
    return StatusCode::SUCCESS;
  } else if(IID_IConcreteSvcType2 == riid) {
    *ppvIF = dynamic_cast<IConcreteSvcType2*> (this);
    return StatusCode::SUCCESS;
  } else {
    return Service::queryInterface(riid, ppvIF);
  }
}

StatusCode SpecificService::initialize() { ... }
StatusCode SpecificService::finalize() { ... }

// Implement the specifics ...
SpecificService::method1() {...}
SpecificService::method2() {...}
SpecificService::method3() {...}
SpecificService::method4() {...}
```


Chapter 12
Tools and ToolSvc

12.1 Overview

Tools are light weight objects whose purpose is to help other components perform their work. A framework service, the `ToolSvc`, is responsible for creating and managing Tools. An `Algorithm` requests the tools it needs to the `ToolSvc`, specifying if requesting a private instance by declaring itself as the parent. Since Tools are managed by the `ToolSvc`, any component[1] can request a tool. Only Algorithms and Services can declare themselves as Tools parents.

In this chapter we first describe these objects and the difference between “private” and “shared” tools. We then look at the `AlgTool` base class and show how to write concrete Tools.

Finally we describe the `ToolSvc` and show how a component can retrieve `Tools` via the service.

12.2 Tools

As mentioned elsewhere `Algorithms` make use of framework services to perform their work. In general the same instance of a service is used by many algorithms and `Services` are setup and initialized once at the beginning of the job by the framework. `Algorithms` also delegate some of their work to sub-algorithms. Creation and execution of sub-algorithms are the responsibilities of the parent algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework while initializing the parent algorithm. The properties of a sub-algorithm are automatically set by the framework but the parent algorithm can change them during execution. Sharing of data between nested algorithms is done via the Transient Event Store.

1. In this chapter we will use an Algorithm as example component requesting tools.

Both Services and Algorithms are created during the initialization stage of a job and live until the jobs ends.

Sometimes an encapsulated piece of code needs to be executed only for specific events, in which case it is desirable to create it only when necessary. On other occasions the same piece of code needs to be executed many times per event. Moreover it can be necessary to execute a sub-algorithm on specific contained objects that are selected by the parent algorithm or have the sub-algorithm produce new contained objects that may or may not be put in the Transient Store. Finally different algorithms may wish to configure the same piece of code slightly differently or share it *as-is* with other algorithms.

To provide this kind of functionality we have introduced a category of processing objects that encapsulate these "light" algorithms. We have called this category Tools.

Some examples of possible tools are single track fitters, association to Monte Carlo truth information, vertexing between particles, smearing of Monte Carlo quantities.

12.2.1 "Private" and "Shared" Tools

Algorithms can share instances of Tools with other Algorithms if the configuration of the tool is suitable. In some cases however an Algorithm will need to customize a tool in a specific way in order to use it. This is possible by requesting the ToolSvc to provide a *"private"* instance of a tool.

If an Algorithm passes a pointer to itself when it asks the ToolSvc to provide it with a tool, it is declaring itself as the parent and a *"private"* instance is supplied. Private instances can be configured according to the needs of each particular Algorithm via jobOptions.

As mentioned above many Algorithms can use a tool *as-is*, in which case only one instance of a Tool is created, configured and passed by the ToolSvc to the different algorithms. This is called a *"shared"* instance. The parent of "shared" tools is the ToolSvc.

12.2.2 The Tool classes

12.2.2.1 The AlgTool base class

The main responsibilities of the AlgTool base class (see Listing 46) are the identification of the tools instances, the initialisation of certain internal pointers when the tool is created and the management of the tools properties. The AlgTool base class also offers some facilities to help in the implementation of derived tools.

Access to Services - A serviceLocator() method is provided to enable the derived tools to locate the services necessary to perform their jobs. Since concrete Tools are instantiated by the ToolSvc upon request, all Services created by the framework prior to the creation of a tool are available. In addition access to the message service is provided via the msgSvc() method. Both pointers are retrieved from the parent of the tool.

Listing 46 The definition of the AlgTool Base class

```
 1:  class AlgTool : public virtual IAlgTool,
 2:                  public virtual IProperty {
 3:
 4:  public:
 5:  // Standard Constructor.
 6:  AlgTool( const std::string& type, const std::string& name,
             const IInterface* parent);
 7:
 8:  virtual const std::string& name() const;
 9:  virtual const std::string& type() const;
10:  virtual const IInterface*  parent() const;
11:
12:  virtual StatusCode setProperty(const Property& p);
13:  virtual StatusCode getProperty(Property* p) const;
14:  virtual const Property& getProperty( const std::string& name) const;
15:  virtual const std::vector<Property*>& getProperties( ) const;
16:
17:  ISvcLocator* serviceLocator();
18:  IMessageSvc* msgSvc();
19:  IMessageSvc* msgSvc() const;
20:
21:  StatusCode setProperties();
22:
23:  StatusCode declareProperty(const std::string& name, int& reference);
24:  StatusCode declareProperty(const std::string& name, double& reference);
25:  StatusCode declareProperty(const std::string& name, bool& reference);
26:  StatusCode declareProperty(const std::string& name,
                                std::string& reference);
27:  StatusCode declareProperty(const std::string& name,
                                std::vector<int>& reference);
28:  StatusCode declareProperty(const std::string& name,
                                std::vector<double>& reference);
29:  StatusCode declareProperty(const std::string& name,
                                std::vector<bool>& reference);
30:  StatusCode declareProperty(const std::string& name,
                                std::vector<std::string>& reference);
31:
32:  protected:
33:    // Standard destructor.
34:    virtual ~AlgTool();
```

Declaring Properties - A set of methods for declaring properties similarly to `Algorithms` is provided. This allows tuning of data members used by the Tools via JobOptions files. The `ToolSvc` takes care of calling the `setProperties()` method of the `AlgTool` base class after having instantiated a tool. Properties need to be declared in the constructor of a `Tool`. The property `outputLevel` is declared in the base class and is identically set to that of the parent component. For details on Properties see section 11.3.1.

Constructor - The base class has a single constructor which takes three arguments. The first is the type (i.e. the class) of the Tool object being instantiated, the second is the full name of the object and the third is a pointer to the `IInterface` of the parent component. The name is used for the identification of the tool instance as described below.The parent interface is used by the tool to access for example the `outputLevel` of the parent.

`IAlgTool` Interface - It consists of three accessor methods for the identification and managment of the tools: `type()`, `name()` and `parent()`. These methods are all implemented by the base class and should not be overridden.

12.2.2.2 Tools identification

A tool instance is identified by its full name. The name consist of the concatenation of the parent name, a dot, and a tool dependent part. The tool dependent part can be specified by the user, when not specified the tool type (i.e. the class) is automatically taken as the tool dependent part of the name. Examples of tool names are `RecPrimaryVertex.VertexSmearer` (a private tool) and `ToolSvc.AddFourMom` (a shared tool). The full name of the tool has to be used in the jobOptions file to set its properties.

12.2.2.3 Concrete tools classes

Operational functionalities of tools must be provided in the derived tool classes. A concrete tool class must inherit directly or indirectly from the `AlgTool` base class to ensure that it has the predefined behaviour needed for management by the `ToolSvc`. The inheritance structure of derived tools is shown in Figure 23. `ConcreteTool1` implements one additional abstract interface while `ConcreteTool3` and `ConcreteTool4` derive from a base class `SubTool` that provides them with additional common functionality.

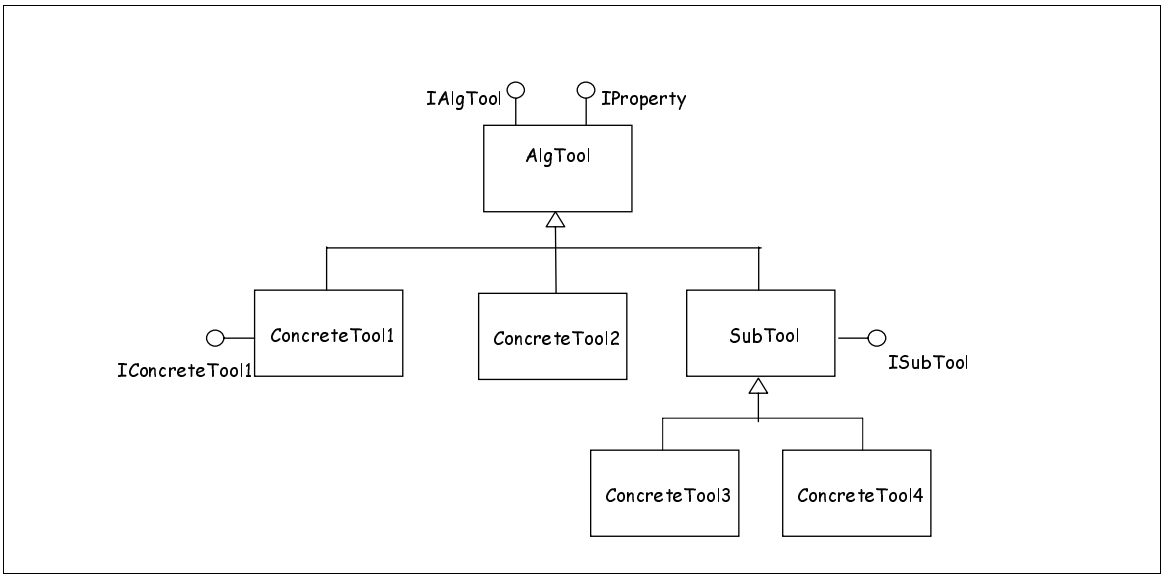


Figure 23 Tools classes hierarchy

The idea is that concrete tools should implement additional interfaces, specific to the task a tool is designed to perform. Specialised tools intended to perform similar tasks can be derived from a common base class that will provide the common functionality and implement the common interface. Consider as example the vertexing of particles, where separate tools can implement different algorithms but the arguments passed are the same.

12.2.2.4 Implementation of concrete tools

An example minimal implementation of a concrete tool is shown in Listing 47 and Listing 48, taken from the `ToolsAnalysis` example application distributed with the Gaudi framework..

Listing 47 Example of a concrete tool minimal implementation header file

```
 1:  #include "Gaudi/Kernel/AlgTool.h"
 2:  class VertexSmearer : public AlgTool {
 3:  public:
 4:    // Constructor
 5:    VertexSmearer( const std::string& type, const std::string& name,
                      const IInterface* parent);
 6:    // Standard Destructor
 7:    virtual ~VertexSmearer() { }
 8:    // specific method of this tool
 1:    StatusCode smear( MyAxVertex* pvertex );
```

Listing 48 Example of a concrete tool minimal implementation file

```
 1:  #include "Gaudi/Kernel/ToolFactory.h"
 2:  // Static factory for instantiation of algtool objects
 3:  static ToolFactory<VertexSmearer> s_factory;
 4:  const IToolFactory& VertexSmearerFactory = s_factory;
 5:
 6:  // Standard Constructor
 7:  VertexSmearer::VertexSmearer(const std::string& type,
                                  const std::string& name,
                                  const IInterface* parent)
                           : AlgTool( type, name, parent ) {
 8:
 9:    // Locate service needed by the specific tool
10:    m_randSvc = 0;
11:    if( serviceLocator() ) {
12:      StatusCode sc=StatusCode::FAILURE;
13:      sc = serviceLocator()->getService( "RndmGenSvc",
                                            IID_IRndmGenSvc,
                                            (IInterface*&) (m_randSvc) );
14:    }
15:    // Declare properties of the specific tool
16:    declareProperty("dxVtx", m_dxVtx = 9 * micrometer);
17:    declareProperty("dyVtx", m_dyVtx = 9 * micrometer);
18:    declareProperty("dzVtx", m_dzVtx = 38 * micrometer);
19:  }
20:  // Implement the specific method ....
21:  StatusCode VertexSmearer::smear( MyAxVertex* pvertex ) {...}
```

The creation of concrete tools is similar to that of Algorithms, making use of a Factory Method. As for Algorithms, Tool factories enable their creator to instantiate new tools without having to include any of the concrete tools header files. A template factory is provided and a tool developer will only need to add the concrete factory in the implementation file as shown in lines 1 to 4 of Listing 48

In addition a concrete tool class must specify a single constructor with the same parameter signatures as the constructor of the `AlgTool` base class as shown in line 5 of Listing 47.

Below is the minimal checklist of the code necessary when developing a Tool:

1. Derive the tool class from the AlgTool base class
2. Provide the constructor
3. Implement the factory adding the lines of code shown in Listing 48

In addition if the tool is implementing an additional interface you may need to:

1. Define specific interface
2. Implement the `queryInterface()` method.
3. Implement the specific interface methods.

12.3 The ToolSvc

The `ToolSvc` manages `Tools`. It is its responsibility to create tools and make them available to `Algorithms` or `Services`.

The `ToolSvc` verifies if a tool type is available and creates the necessary instance after having verified if it doesn't already exist. If a tool instance exists the `ToolSvc` will not create a new identical one but pass to the algorithm the existing instance. Tools are created on a "first request" basis: the first Algorithm requesting a tool will prompt its creation. The relationship between an algorithm, the ToolSvc and Tools is shown in Figure 24.

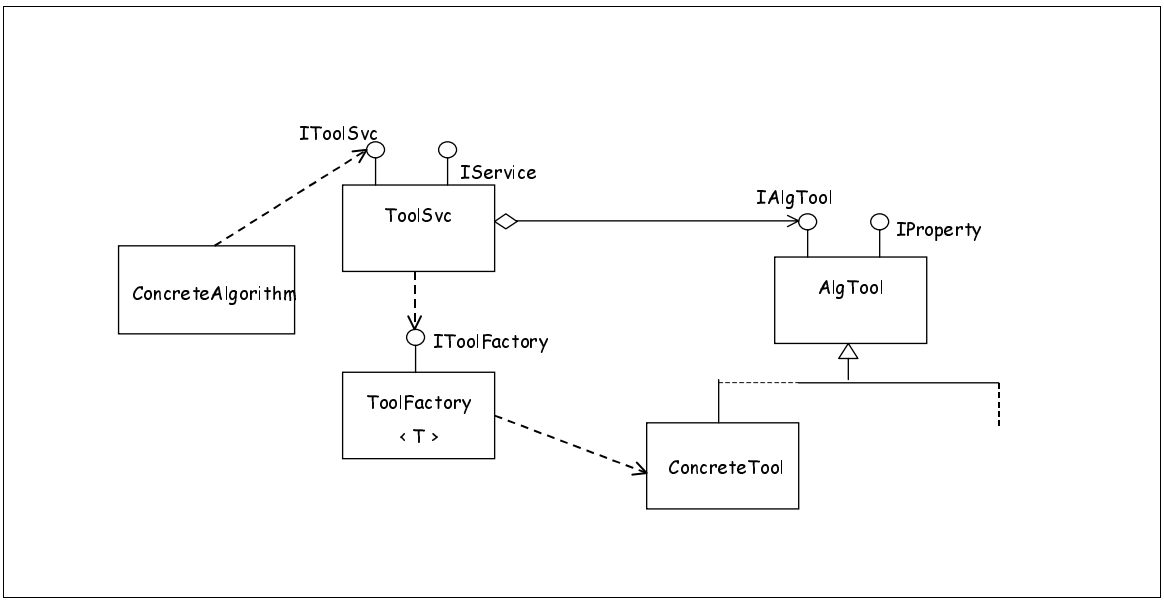


Figure 24 ToolSvc design diagram

The `ToolSvc` will "hold" a tool until it is no longer used by any component or until the `finalize()` method of the tool service is called. Algorithms can inform the `ToolSvc` they are not going to use a tool previously requested via the `releaseTool` method of the `IToolSvc` interface[1].

At the moment the ToolSvc is provided as an external service and has to be requested via the jobOptions file using the property ApplicationMgr.ExtSvc as shown below:

```
ApplicationMgr.ExtSvc += { ToolSvc };
```

Algorithms wishing to use the service need to locate it. This can be done using the serviceLocator() accessor method provided in the Algorithm base class that will return the IToolSvc interface as shown in the lines below.

```
include "Gaudi/Interfaces/IToolSvc.h"
...
IToolSvc* toolSvc=0;
StatusCode sc = serviceLocator()->getService( "ToolSvc",
                                              IID_IToolSvc,
                                                   (IInterface*&)(toolSvc));

if ( sc.isFailure) {
...
```

In future releases, with the consolidation of Tools and ToolSvc, the service will become one of those created by default by the Application manager and the Algorithm base class will have an accessor method to the ToolSvc.

12.3.1 Retrieval of tools via the IToolSvc interface

The IToolSvc interface is the ToolSvc specific interface providing methods to retrieve tools. The interface has two retrieve methods that differ in their parameters signature, as shown in Listing 49

Listing 49 The IToolSvc interface methods

```
1:  virtual StatusCode retrieve(const std::string& type,
                                IAlgTool*& tool,
                                const IInterface* parent=0,
                                bool createIf=true ) = 0;
2:  virtual StatusCode retrieve(const std::string& type,
                                const std::string& name,
                                IAlgTool*& tool,
                                const IInterface* parent=0,
                                bool createIf=true ) = 0;
```

The arguments of the method shown in Listing 49, line 1, are the tool type (i.e. the class) and the IAlgTool interface of the returned tool. In addition there are two arguments with default values: one is the IInterface of the component requesting the tool, the other a boolean creation flag. If the component requesting a tool passes a pointer to itself as the third argument, it declares to the ToolSvc that is asking a “private” instance of the tool. By default

1. The releaseTool method is not available in this release

a “shared” instance is provided. In general if the requested instance of a Tool does not exist the ToolSvc will create it. This behaviour can be changed by setting to `false` the last argument of the method.

The method shown in Listing 49, line 2 differs from the one shown in line 1 by an extra argument, a string specifying the tool dependent part of the full tool `name`. This enables a component to request two separately configurable instances of the same tool.

To help the retrieval of concrete tools two template functions, as shown in Listing 50, are provided in the `IToolSvc` interface file.

Listing 50 The IToolSvc template methods

```
1:  template <class T>
2:    StatusCode retrieveTool( const std::string& type,
                               T*& tool,
                               const IInterface* parent=0,
                               bool createIf=true ) {...}
3:  template <class T>
4:    StatusCode retrieveTool( const std::string& type,
                               const std::string& name,
                               T*& tool,
                               const IInterface* parent=0,
                               bool createIf=true ) {...}
```

The two template methods correspond to the `IToolSvc` retrieve methods but have the tool returned as a template parameter. Using these methods the component retrieving a tool avoids explicit dynamic-casting to specific additional interfaces or to derived classes.

Listing 51 shows an example of retrieval of a shared and of a common tool.

Listing 51 Example of retrieval of a shared tool in line 8: and of a private tool in line 14:

```
 1:  IToolSvc* toolsvc = 0;
 2:  sc = serviceLocator()->getService( "ToolSvc", IID_IToolSvc,
                                        (IInterface*&)( toolsvc ));
 3:  if( sc.isFailure() ) {
 4:    log << MSG::FATAL << "    Unable to locate Tool Service" << endreq;
       return sc;
 5:  }
 6:  // Example of tool belonging to the ToolSvc and shared between
 7:  // algorithms
 8:  sc = toolsvc->retrieveTool("AddFourMom", m_sum4p );
 9:  if( sc.isFailure() ) {
10:    log << MSG::FATAL << "    Unable to create AddFourMom tool" << endreq;
11:    return sc;
12:  }
13:  // Example of private tool
14:  sc = toolsvc->retrieveTool("ImpactPar", m_ip, this );
15:  if( sc.isFailure() ) {
16:    log << MSG::FATAL << "    Unable to create ImpactPar tool" << endreq;
17:    return sc;
18:  }
```


Chapter 13
Converters

13.1 Overview

Consider a small piece of the LHCb detector; a silicon wafer for example. This “object” will appear in many contexts: it may be drawn in an event display, it may be traversed by particles in a Geant4 simulation, its position and orientation may be stored in a database, the layout of its strips may be queried in an analysis program, etc. All of these uses or views of the silicon wafer will require code.

How to encompass the need for these different views within Gaudi was one of the key issues in the design of the framework. In this chapter we outline the design adopted for the framework and look at how the conversion process works. This is followed by two sections which deal with the technicalities of writing converters for reading SicB data and for reading from and writing to ROOT files.

13.2 Persistency converters

Gaudi gives the possibility to read in event data either from Zebra or from ROOT files and to write data back to disk in ROOT files. This data may then of course be read again at a later date. Figure 25 is a schematic illustrating how converters fit into the transient-persistent translation of event data. We will not discuss in detail how the transient data store (e.g. the event data service) or the persistency service work, but simply look at the flow of data in order to understand how converters are used.

One of the issues considered when designing the Gaudi framework was the capability for users to “create their own data types and save objects of those types along with references to already existing objects”. A related issue was the possibility of having links between objects which reside in different stores (i.e. files and databases) and even between objects in different types of store.

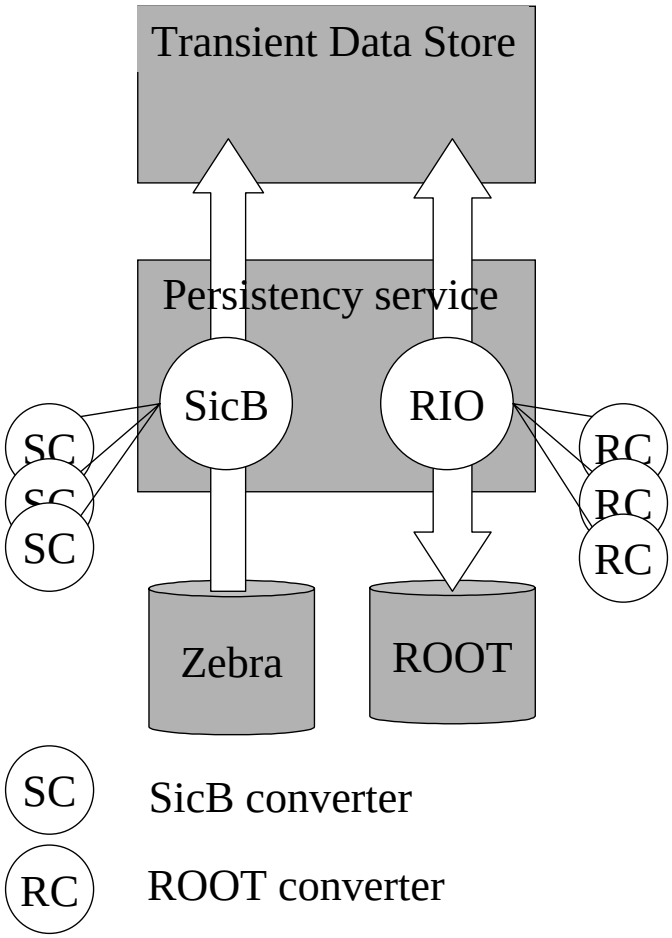


Figure 25 Persistency conversion services in Gaudi

The Gaudi framework gives the possibility to save data objects into ROOT based files. Thus, since our principal store of data for the moment is still SicB data in Zebra files we see immediately an application of the issues mentioned above. Figure 25. shows that data may be read from SicB files and ROOT files into the transient event data store and that data may be written into ROOT files. It is the job of the persistency service to orchestrate this transfer of data between memory and disk.

The figure shows two “slave” services: the SicB conversion service and the RIO (ROOT I/O) service. These services are responsible for managing the conversion of objects between their transient and persistent representations. Each one has a number of converter objects which are actually responsible for the conversion itself. As illustrated by the figure a particular converter object converts between the transient representation and one other form, here either Zebra or ROOT.

13.3 Collaborators in the conversion process

In general the conversion process occurs between the transient representation of an object and some other representation. In this chapter we will be using persistent forms, but it should be borne in mind that this could be any other “transient” form such as those required for visualisation or those which serve as input into other packages (e.g. Geant4).

Figure 26 shows the interfaces (classes with names beginning in I) which must be

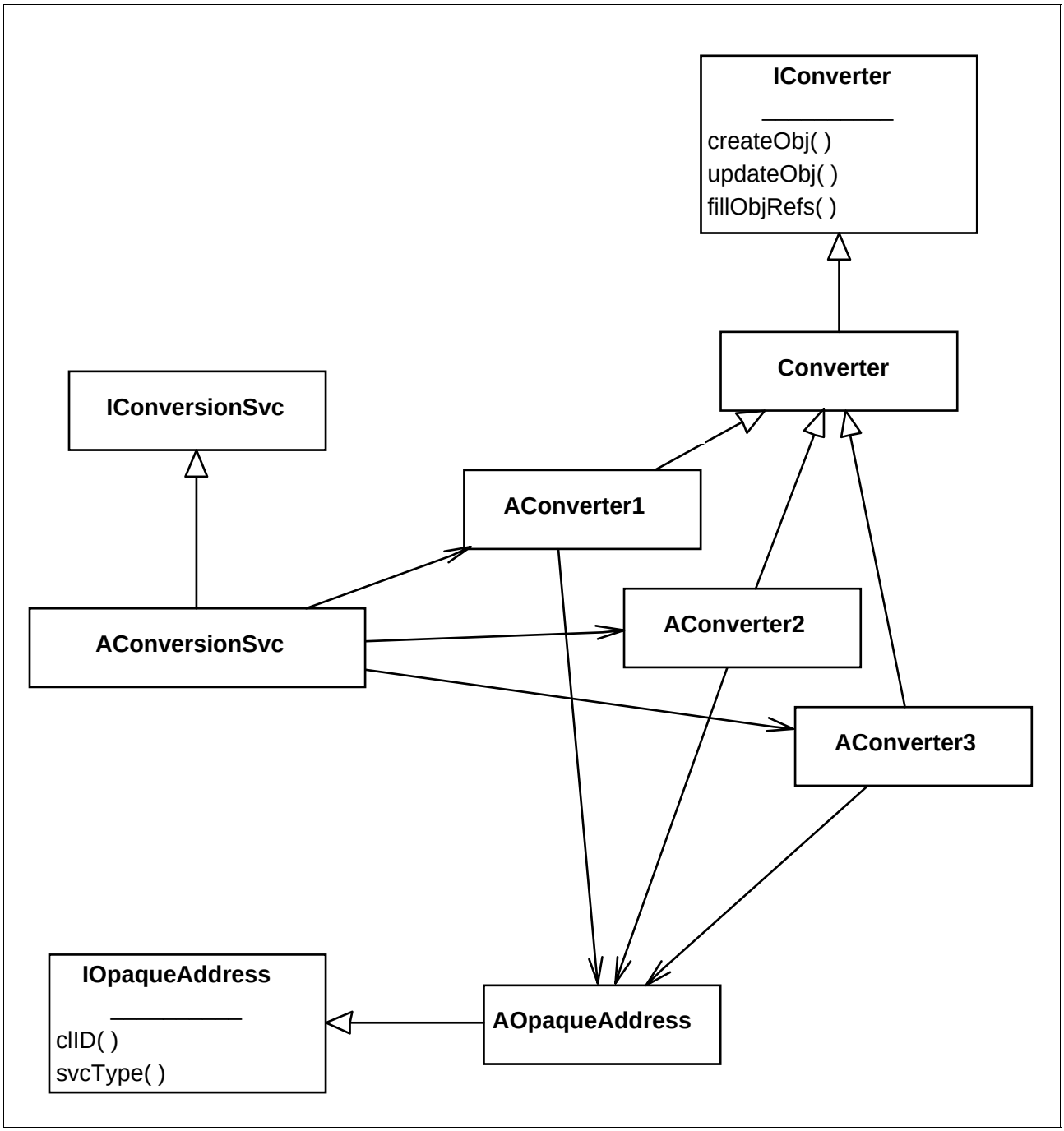


Figure 26 The classes (and interfaces) collaborating in the conversion process.

implemented in order for the conversion process to function. The conversion process is essentially a collaboration between the following types:

- `IConversionSvc`
- `IConverter`
- `IOpaqueAddress`

For each persistent technology, or “non-transient” representation, a specific conversion service is required. This is illustrated in the figure by the class `AConversionSvc` which implements the `IConversionSvc` interface.

A given conversion service will have at its disposal a set of converters. These converters are both type and technology specific. In other words a converter knows how to convert a single transient type (e.g. `MuonHit`) into a single persistent type (e.g. `RootMuonHit`) and vice versa. Specific converters implement the `IConverter` interface, possibly by extending an existing converter base class.

A third collaborator in this process are the opaque address objects. A concrete opaque address class must implement the `IOpaqueAddress` interface. This interface allows the address to be passed around between the transient data service, the persistency service, and the conversion services without any of them being able to actually decode the address. Opaque address objects are also technology specific. The internals of a `SicBAddress` object are different from those of a `RootAddress` object.

Only the converters themselves know how to decode an opaque address. In other words only converters are permitted to invoke those methods of an opaque address object which do not form a part of the `IOpaqueAddress` interface.

Converter objects must be “registered” with the conversion service in order to be usable. For the “standard” converters this will be done automatically. For user defined converters (for user defined types) this registration must be done at initialisation time (see Chapter 6).

13.4 The conversion process

As an example (see Figure 27) we consider a request from the event data service to the persistency service for an object to be loaded from a data file.

As we saw previously, the persistency service has one conversion service slave for each persistent technology in use. The persistency service receives the request in the form of an opaque address object. In order to decide which conversion service the request should be passed onto the `svcType()` method of the `IOpaqueAddress` interface is invoked. This returns a “technology identifier” which allows the persistency service to choose a conversion service.

The request to load an object (or objects) is then passed onto a specific conversion service. This service then invokes another method of the `IOpaqueAddress` interface, `clID()`, in order to decide which converter will actually perform the conversion. The opaque address is then passed onto the concrete converter who knows how to decode it and create the appropriate transient object.

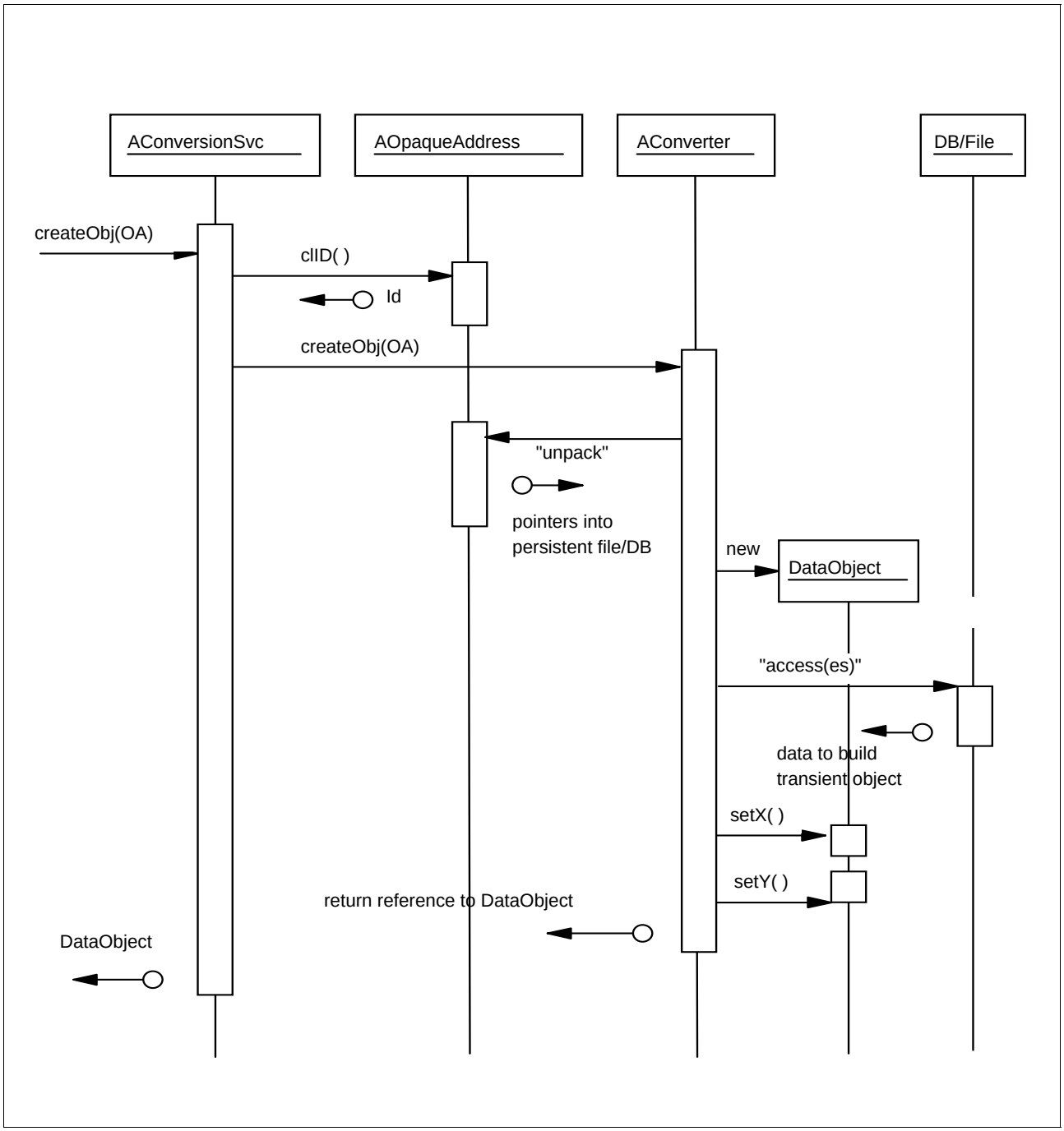


Figure 27 A trace of the creation of a new transient object.

The converter is specific to a specific type, thus it may immediately create an object of that type with the new operator. The converter must now “unpack” the opaque address, i.e. make use of accessor methods specific to the address type in order to get the necessary information from the persistent store.

For example, a SicB converter might get the name of a bank from the address and use that to locate the required information in the SicB common block. On the other hand a ROOT converter may extract a file name, the names of a ROOT `TTree` and an index from the address and use these to load an object from a ROOT file. The converter would then use the accessor

methods of this “persistent” object in order to extract the information necessary to build the transient object.

We can see that the detailed steps performed within a converter depend very much on the nature of the non-transient data and (to a lesser extent) on the type of the object being built.

If all transient objects were independent, i.e. if there were no references between objects then the job would be finished. However in general objects in the transient store do contain references to other objects.

These references can be of two kinds:

- “Macroscopic” references appear as separate “leafs” in the data store. They have to be registered with a separate opaque address structure in the data directory of the object being converted . This must be done after the object was registered in the data store in the method `fillObjRefs()`.
- Internal references must be handled differently. There are two possibilities for resolving internal references:
    - Load on demand: If the object the reference points to should only be loaded when accessed, the pointer must no longer be a raw C++ pointer, but rather a smart pointer object containing itself the information for later resolution of the reference. This is the preferred solution for references to objects within the same data store, e.g. references from the Monte-Carlo tracks to the Monte-Carlo vertices. Please see in the corresponding SicB converter implementations how to construct these smart pointer objects. Late loading is highly preferable compared to the second possibility.
    - Filling of raw C++ pointers: Here things are a little more complicated and introduces the need for a second step in the process. This is only necessary if the object points to an object in another store, e.g. the detector data store. To resolve the reference a converter has to retrieve the other object and set the raw pointer. These references should be set in the `fillObjRefs()` method. This of cousre is more complicated, because it must be ensured that both objects are present at the time the reference is accessed (i.e. when the pointer is actually used).

13.5 Converter implementation - general considerations

After covering the ground work in the preceding sections, let us look exactly what needs to be implemented in a specific converter class. The starting point is the `Converter` base class from which a user converter should be derived. For concreteness let us partially develop a converter for the `UDO` class of Chapter 6.

The converter shown in Listing 52 is responsible for the conversion of `UDO` type objects into objects that may be stored into an Objectivity database and vice-versa. The `UDOCnv` constructor calls the Converter base class constructor with two arguments which contain this information. These are the values `CLID_UDO`, defined in the `UDO` class, and `Objectivity_StorageType` which is also defined elsewhere. The first two `extern` statements simply state that these two identifiers are defined elsewhere.

Listing 52 An example converter class

```
// Converter for class UDO.
extern const CLID& CLID_UDO;
extern unsigned char OBJY_StorageType;

static CnvFactory<UDOCnv> s_factory;
const ICnvFactory& UDOCnvFactory = s_factory;

class UDOCnv : public Converter {
public:
  UDOCnv(ISvcLocator* svcLoc) :
      Converter(Objectivity_StorageType, CLID_UDO, svcLoc) { }

  createRep(DataObject* pO, IOpaqueAddress*& a);
  createObj(IOpaqueAddress* pa, DataObject*& pO);

  fillObjRefs( ... );
  fillRepRefs( ... );
}
```

All of the “book-keeping” can now be done by the `Converter` base class. It only remains to fill in the guts of the converter. If objects of type `UDO` have no links to other objects, then it suffices to implement the methods `createRep()` for conversion from the transient form (to Objectivity in this case) and `createObj()` for the conversion to the transient form.

If the object contains links to other objects then it is also necessary to implement the methods `fillRepRefs()` and `fillObjRefs()`.

13.6 Storing Data using the ROOT I/O Engine

One possibility for storing data is to use the ROOT I/O engine to write ROOT files. Although ROOT by itself is not an object oriented database, with modest effort a structure can be built on top to allow the Converters to emulate this behaviour. In particular, the issue of object linking had to be solved in order to resolve pointers in the transient world.

The concept of ROOT supporting paged tuples called trees and branches is adequate for storing bulk event data. Trees split into one or several branches containing individual leaves with data. The data structure within the Gaudi data store is tree like (see Figure 28).

In the transient world Gaudi objects are sub class instances of the “DataObject”. The DataObject offers some basic functionality like the implicit data directory which allows e.g. to browse a data store. This tree structure will be mapped to a flat structure in the ROOT file resulting in a separate tree representing each leaf of the data store. Each data tree contains a single branch containing objects of the same type. The Gaudi tree is split up into individual ROOT trees in order to give easy access to individual items represented in the transient model without the need of loading complete events from the root file i.e. to allow for selective data retrieval. The feature of ROOT supporting selective data reading using split trees seemed not to be too attractive since generally complete nodes in the transient store should be made available in one go.

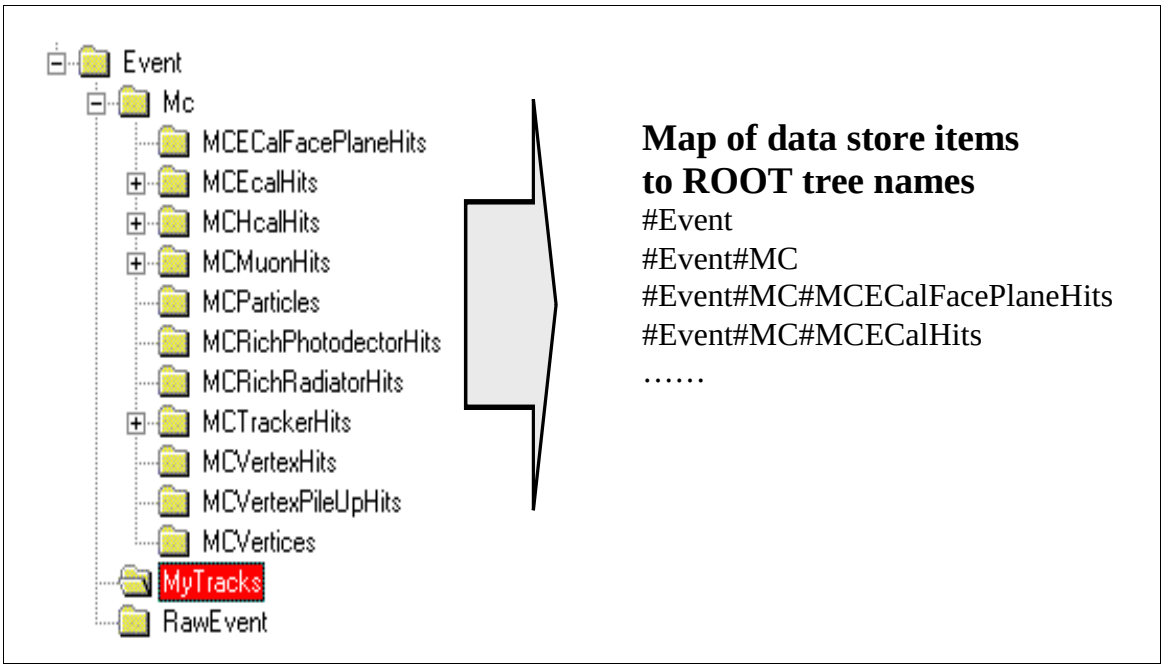


Figure 28 The Transient data store and its mapping in the Root file. Note that the "/" used within the data store to identify separate layers are converted to "#" since the "/" within ROOT denominates directory entries

However, ROOT expects "ROOT" objects, they must inherit from `TObject`. Therefore the objects from the transient store have to be converted to objects understandable by ROOT.

The following sections are an introduction to the machinery provided by the Gaudi framework to achieve the migration of transient objects to persistent objects. The ROOT specific aspects are not discussed here; the documentation of the ROOT I/O engine can be found at the ROOT web site *http://root.cern.ch*). Note that Gaudi only uses the I/O engine, not all ROOT classes are available.

Within Gaudi the ROOT I/O engine is implemented in the DbCnv package.

13.7 The Conversion from Transient Objects to ROOT Objects

As for any conversion of data from one representation to another within the Gaudi framework, conversion to/from ROOT objects is based on Converters. The support of a "generic" Converter accesses pre-defined entry points in each object. The transient object converts itself to an abstract byte stream.

However, for specialized objects specific converters can be built by virtual overrides of the base class.

Whenever objects must change their representation within Gaudi, data converters are involved. For the ROOT case the converters must have some knowledge of ROOT internals and the service finally used to migrate ROOT objects (->`TObject`) to a file. In the same way the converter must be able to translate the functionality of the `DataObject` component

to / from the Root storage. Within ROOT itself the object is stored as a Binary Large Object (BLOB).

The instantiation of the appropriate converter is done by a macro. The macro instantiates also the converter factory used to instantiate the requested converter. Hence, all other user code is shielded from the implementation and definitions of the ROOT specific code.

Listing 53 Implementing a “generic” converter for the transient class *Event*.

```
 1: // Include files
 2: #include "LHCbEvent/TopLevel/ObjectVector.h"
 3: #include "LHCbEvent/TopLevel/ObjectList.h"
 4: #include "DbCnv/DbGenericConverter.h"
 5: // Converter implementation for objects of class Event
 6: #include "LHCbEvent/TopLevel/Event.h"
 7: _ImplementConverter(Event)
```

The macro needs a few words of explanation: the instantiated converters are able to create transient objects of type `Event`. The corresponding persistent type is of a generic type, the data are stored as a machine independent byte stream. It is mandatory that the Event class implements a streamer method “`serialize`”. An example of the Event class is shown in Listing 54.

The instantiated converter is of the type `DbGenericConverter` and the instance of the instantiating factory has the instance name `DbEventCnvFactory.`.

Listing 54 Serialisation of the class *Event*.

```
 1:  /// Serialize the object for writing
 2: virtual StreamBuffer& serialize( StreamBuffer& s ) const              {
 3:     DataObject::serialize(s);
 4:     return s
 5:       << m_event
 6:       << m_run
 7:       << m_time;
 8: }
 9: /// Serialize the object for reading
10: virtual StreamBuffer& serialize( StreamBuffer& s )                    {
11:     DataObject::serialize(s);
12:     return s
13:       >> m_event
14:       >> m_run
15:       >> m_time;
16: }
```

13.7.1 Non Identifiable Objects

Non identifiable objects cannot directly be retrieved / stored from the data store. Usually they are small and in any case they are contained by a container object. Examples are particles (class `MCParticle`), hits (class `MCHitBase` and others) or vertices (class `MCVertex`). These classes can be converted using a generic container converter. Container converters exist currently for lists and vectors. The containers rely on the serialize mothods of the contained

objects. The serialisation is able to understand smart references to other objects within the same data store: e.g. the reference from the `MCParticle` to the `MCVertex`. Listing 55 shows an example of the serialize methods of the `MCParticle` class

Listing 55 Serialisation of the class *Event*..

```
 1: /// Serialize the object for writing
 2: inline StreamBuffer& MCParticle::serialize( StreamBuffer& s ) const {
 3:   ContainedObject::serialize(s);
 4:   unsigned char u = (m_oscillationFlag) ? 1 : 0;
 5:   return s
 6:     << m_fourMomentum
 7:     << m_particleID
 8:     << m_flavourHistory
 9:     << u
10:     << m_originMCVertex(this)    // Stream a reference to another object
11:     << m_decayMCVertices(this);  // Stream a vector of references
12: }
13:
14:
15: /// Serialize the object for reading
16: inline StreamBuffer& MCParticle::serialize( StreamBuffer& s ) {
17:   ContainedObject::serialize(s);
18:   unsigned char u;
19:   s >> m_fourMomentum
20:     >> m_particleID
21:     >> m_flavourHistory
22:     >> u
23:     >> m_originMCVertex(this)    // Stream a reference to another object
24:     >> m_decayMCVertices(this);  // Stream a vector of references
25:   m_oscillationFlag = (u) ? true : false;
26:   return s;
27: }
```

Please refer to the Gaudi example Rio.Example1 for further details how to store objects in ROOT files.

13.8 Storing Data using other I/O Engines

Once objects are stored as BLOBs, it is possible to adopt any storage technology supporting this datatype. This is the case not only for ROOT, but also for

- Objectivity / DB
- most relational databases, which support an ODBC interface like
    - Microsoft Access,
    - Microsoft SQL Server,
    - MySQL,
    - ORACLE and others.

Note that although storing objects using these technologies is possible, there is currently no experiment wide policy on how to use Objectivity or other client server based technologies. For this reason only the example to store data using Microsoft Access is described in the example Rio.Example1. All other technologies are currenly not supported. If you desperately want to use SQL Server, MySQL or Objectivity, please contact Markus Frank (Markus.Frank@cern.ch).

Chapter 14
Accessing SICB facilities

14.1 Overview

In order to facilitate the transition towards C++ based code we have implemented a number of features into the Gaudi framework whose purpose is to allow access to facilities existing within SICB but not yet existing within the framework itself. Gaudi also can read data produced with SICBMC or SICBDST.

In this chapter we cover: staging data from DSTs, converting SICB data for use by Gaudi algorithms, accessing the magnetic field map, accessing the SICB geometry description and the use of the SUINIT, SUANAL and SULAST routines from within the Gaudi framework.

When using the geometry and magnetic field descriptions described here, you should not forget that they are a temporary solution and that they will disappear at some point in the future. Gaudi includes already the machinery to provide any algorithm with the detector data stored in XML format. Investing some time now to describe your detector in XML may be, in many cases, more convenient than using the old SICB routines to access detector description data. If, for any reason, you have to use the old tools, use them only to populate some pre-defined class which can then be used in your algorithms. In this way, when you decide to move to the new detector description tools in Gaudi, the required changes in your code will be confined to the parts which access the geometry.

14.2 Reading tapes

Important Note: Accessing tapes and the following instructions may only work in the CERN installation. Due to a problem with the RFIO software needed to access data on the shift staging disk pools, only the first method of the methods described below works on NT.

There are three ways to specify an input event data file in Gaudi:

1. Read one or more files on disk by setting the appropriate option of the application manager in the job options file:

```
ApplicationMgr.EvtSel = "FILE
$AFSROOT\cern.ch\lhcb\data\mc\sicb_bpipi_v233_100ev.dst1,
$AFSROOT\cern.ch\lhcb\data\mc\\sicb_mbias_v233_10ev.dst2";
```

2. Specify a data set by giving the job identification numbers in the book-keeping data base, as was done in SICB. The framework will query the ORACLE database in order to find the corresponding tape and then stage it in. The access to the database is through the oraweb01 web server at CERN, as for the `mcstagein` script.

   The job IDs are specified in the job options file as follows:

```
ApplicationMgr.EvtSel = "JOBID 11758, 11759";
```

3. Read one or more tape files, even if they are not in the LHCb production book-keeping database, for example test beam data. The user should tell the Event Selector which tape(s) and file(s) to read in the following way:

```
ApplicationMgr.EvtSel = "TAPE Y21221-7, Y21223-24";
```

   The format is <Volume serial number of the tape>-<File sequence number>

When a Gaudi job requires to stage more than one tape the programs waits for the first tape to be staged. The rest of the required tapes are staged while the first file is processed. One the program ends reading the first file it will check if the next file is ready before continuing. It the tape is not staged yet the program writes a message and waits until the file is in the staging disk pools.

Skipping events When reading SICB data, one may wish to skip some events. This can be done by setting the property `EventSelector.FirstEvent`. For eaxmple, to start processing event 109, add the following job option::

```
EventSelector.FirstEvent = 109;
```

The algorithms will run only after reading that event. Note that Gaudi will take few seconds to reach the requested event as it has to read all the records in the Zebra file before the one requested.

14.3 Populating the GAUDI transient data store: SICB Converters

14.3.1 General considerations

Access to the SICB data sets is basically via wrappers to the Fortran code. A complete event is read into the Zebra common block, and then the conversion to transient objects is done on request by specialised converters.

As mentioned in Chapter 13, a converter must implement the `IConverter` interface, by deriving from a specific base class. In this way any actions which are in common to all converters of a specific technology may be implemented in a single place.

In the following section we give detailed instructions on how to implement converters within the SicbCnv package. These are intended primarily for Gaudi developers themselves.

14.3.2 Implementing converters in the SicbCnv package

SICB converters are available currently only for reading, writing back into persistent storage (ZEBRA files) is not possible at present. Typically GAUDI DataObjects can be of two types:

- Simple classes, which contain the data of a single SICB bank. These classes are of type DataObject. An example is the `Event` class containing the data of the PASS bank.
- Container classes, which contain data from multiple SICB banks. An example is the `ObjectVector<MCParticle>`, which contains Monte-Carlo particles with data from the ATMC bank.

For both types of converters template files exist, which should ease the creation of user converters:

- `SicbCnv.Class.Template.cpp` and `SicbCnv.Class.Template.h` to be used when writing a converter for a single class.
- `SicbCnv.Class.Container.cpp` and `SicbCnv.Container.Template.h` to be used when writing a container of an object container.

The template files are located in the `$LHCBSOFTSicbCnv/<version>/doc` directory. If you intend to write your own SICB converter, use the templates and follow the following instructions:

- Copy `SicbCnv.xxxx.Template.h` to `Sicb<your-class>Cnv.h`, where `<your-class>` is the name of your persistent class.
- Copy `SicbCnv.xxxx.Template.cpp` to `Sicb<your-class>Cnv.cpp`
- Now customize the header and the implementation file
    - Follow `TODO` instructions in `Sicb<your-class>Cnv.h`
    - Follow `TODO` instructions in `Sicb<your-class>Cnv.cpp`

- The converter factory must be made known to the system. This in fact depends on the linking mechanism: If the converter is linked into the executable as an object file, no action is necessary. However, usually the converter code resides in a shared or archive library. In this case the library must have an initialisation routine which creates an artificial reference to the created converter and forces the linker to include the code in the executable. An example of creating such a reference can be found in the file `$LHCBSOFTSicbCnv/<version>/SicbCnv/SicbCnvDll/SicbCnv_load.cpp`. The convention for these initialization files is the following: for any other package replace the string "`SicbCnv`" with "`OtherPackage`".
- Compile link, debug
- Once the converter works, remove unnecessary `TODO` comments.

14.4 Access to the Magnetic Field

The magnetic field map will be accessible in the future via the transient detector store. For the time being, as this is not implemented and as access to the magnetic field has been requested, we have provided a magnetic field service. Again this is effectively just a wrapper which uses SICB routines to read the information from a `.cdf` file.

The location of the `field.cdf` file is provided by the `standard.stream` file which is read in by a SICB routine called from Gaudi. This file is in the data base area, `$LHCBDBASE/standard.stream` in AFS. For every version the file used in the production is read in. The location of the `standard.stream` file will be taken from an environment variable as per normal SICB operation.

To use the Magnetic field service one should modify the `jobOptions.txt` file to include the following:

```
ApplicationMgr.ExtSvc = { "MagneticFieldSvc"};
```

Any algorithm which requires the use of the service makes a request via the `serviceLocator()` method of the `Algorithm` base class:

```
IMagneticFieldSvc* pIMF= 0;
serviceLocator()->getService("MagneticFieldSvc",
                             IID_IMagneticFieldSvc,
                             reinterpret_cast<IInterface*&>( pIMF) );
```

The service provides a method:

```
StatusCode fieldVector(HepPoint3D& Pos, HepVector3D& field)
```


which gives a magnetic field vector at a given point in space, for example:

```
HepPoint3D P(10.*cm, 10.*cm, 120.*cm);
HepVector3D B;
pIMF->fieldVector( P, B );
```

The magnetic field service uses a new version of the SICB routine `GUFLD`. In this new version the best possible description of the magnet geometry and the field are assumed in order to eliminate dependencies with other parts of SICB. Technically in SICB this corresponds to:

```
IMAGLEV = IUVERS('GEOM','MAGN') = 4
IFLDLEV = IUVERS('GEOM','MFLD') = 4
```

These two parameters have been fixed to 4 in the production since a few months before the Technical Proposal: version 111 of SICB. Thus this seems to be a reasonable restriction until the field map is provided in the detector description database.

For an example of the use of this service see the sub-algorithm `readMagField` in the example `FieldGeom` distributed with the release.

14.5 Accessing the SICB detector geometry from Gaudi

As discussed previously, the detector geometry will be included along with the field map in the XML detector description database. Currently only a part of the LHCb detector is in the new datbase. However, the detector geometry used in the production of the data can be accessed by calling a function in the `SicbFortran` name space (this function is just a wrapper for the FORTRAN function of similar name):

```
void SicbFortran::utdget(const std::string& a1, const std::string& a2,
                         int& nParam, int* data);
```

`nParam` should be set to the number of data words required and on return from the function will contain the number of data words actually copied into the array: `data`. The first string contains the name of the sub detector whose geometry is being requested and the second string is a list of options:

'V'- version;
'G' - geometry description parameters (default);
'C' - calculated geometry (not in *.cdf);
'H' - hit description parameters;
'D' - digitization parameters;
'R' - reconstruction parameters;
'F' - floating point parameters (default);
'I' - integer parameters;
'N' - take parameters from the *.cdf file

'L' - ZEBRA pointer to the beginning of the parameters storage is returned in IARRAY(1)

An algorithm requiring this access should include the header file:

```
#include "SicbCnv/TopLevel/SicbFortran.h"
```

and call it so:

```
float rpar[300];

SicbFortran::utdget("WDRF","D",mpar, (int *) rpar);
log << MSG::INFO << " wdpar(" << j << ") = " << vf[j] << endreq;
```

Note that the data returned by the function is written into an integer array. However we can also read floating point numbers as in the code fragment above by casting a float array.

One should notice that the geometry returned by this function is that which was used in the production of the data and not that which is in the current version of the `.cdf` files. Only if the option 'N' is specified are the `.cdf` files read in from the standard location. In order to be able to use the array of parameters returned one has to know in advance the organization of these data in the cdf file since the data are stored in an array and not in a common block with named variables.

The sub-algorithm `readTRackerGeom` in the example `FieldGeom` extracts and writes out some digitization and geometry parameters of the outer tracker.

14.6 Using fortran code in Gaudi

Existing FORTRAN code can be used within a Gaudi application by using the `FortranAlgorithm` class.This is a standard Gaudi algorithm which calls the FORTRAN routines: `SUINIT` in the `initialize()` method, `SUANAL` in the `execute()` method for each event, and `SULAST` in `finalize()`. Implementing these three routines allows you to write code in FORTRAN and have it called from within Gaudi, in particular to import routines already written in SICB.

Note, however that there are some points that should be kept in mind when importing SICB code into Gaudi. The following list is far from being complete but we hope that it will help anyone using the `FortranAlgorithm`. It may be updated in the future with the user's experiences and our own findings.

- Gaudi is linked with only two sub-packages of SICB: `Finclude` and `Futio`. This means that taking some code from SICB and running it successfully in Gaudi will not always be straight forward. Every case will have to be studied separately. In some cases you may find that the effort required to make the code compatible with Gaudi

is comparable to writing it directly in C++. For some pieces of code the integration into Gaudi may be very simple. The difficulties will come mainly from dependencies of the code you want to include on other parts of SICB. For instance there may be common blocks which your code needs but which are never initialized.

- As most of SICB is not executed, you will have either to include and execute the required initialization or to try to eliminate those dependencies.
- Gaudi uses SICB to access the data in the DST's, to extract the information from the banks and to convert the data into the appropriate classes. This needs some initialization but not every SICB initialization step is followed. The standard.stream file and setup.cdf are read in from the standard locations. The program will have access to the data in the cdf files which were used to produce the DST.
- Sicb.dat is not read in. If you need to change the running conditions of your program do it using the jobOptions.txt file or write your own cards file and read it in SUINIT.
- In `Finclude` you will find all the NAME_BANK.INC and NAME_FUNC.INC include files. This means that you have access to the bank data with the usual utilities provided by SICB. For example, to access the momentum of the reconstructed track one can use (as in SICB):

```
P = AXTK$P(NOBJ)
```

- `Futio` includes most of the UT* and UB routines which can therefore be used by fortran code within Gaudi.
- Initialize the histogram files yourself in SUINIT. Gaudi initializes the histogram service but this can be accessed only from C++ code.

14.7 Handling pile up in Gaudi.

In this section we explain the pile-up structure implemented in the Gaudi framework.

Pile-up in Gaudi is performed by a pile-up algorithm. An example (PileUpAlg) can be found in the package SicbCnv. The pile-up algorithm creates a second instance of the event selector which has to be configured in the job options file. The leading event will be read in from the EventSelector created by the ApplicationMgr. The piled-up events are read in from the second instance created by the pile-up algorithm. There are two different iterators, one for each EventSelector instance, which loop over these two data streams.

When reading the ZEBRA files produced by SICB the events are merged at the level of the ZEBRA common blocks by a call to the Futio routine REMERGE. Every C++ algorithm requesting data from one of the converted banks will get the merged data. Every FORTRAN algorithm using the SICB banks will also read the merged data as SICBDST does.

`PileUpAlg` must be the first algorithm called, other algorithms will access the merged events when they retrieve some data from the store. The pile-up algorithm controls the number of events which have to be piled-up to every leading event. `PileUpAlg` uses a C++ version of the SICB routine RELUMI to get the number of pile-up events as a function of the luminosity and some other beam parameters. Those parameters are currently read in from the beam.cdf

file. The C++ version of RELUMI uses the random number service in Gaudi. Other implementations of the algorithm, for instance to return a fix number of pile-up events every time, may be implemented if they are needed.

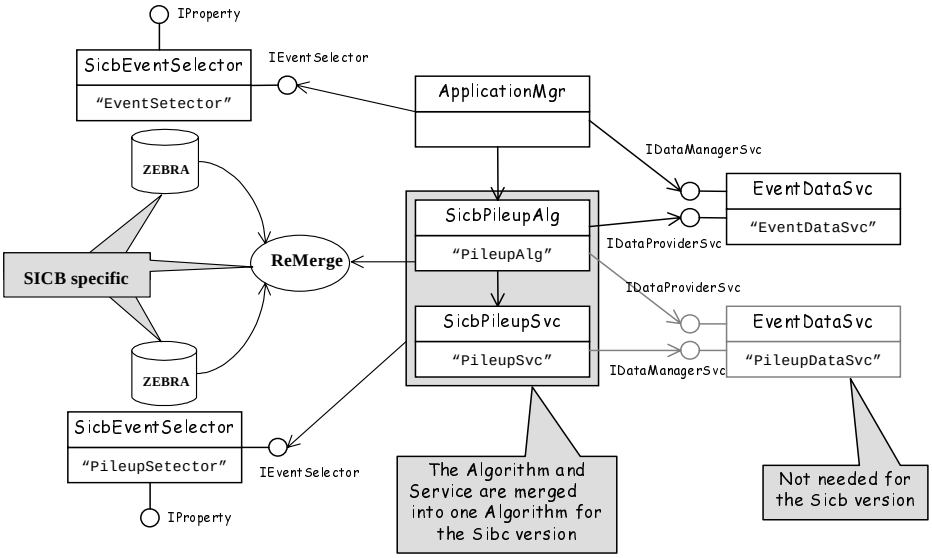


Figure 29 Pile-up in Gaudi

In the future, when data are not stored in ZEBRA files but in some other data base, both event selectors will use two different data services to access different data stores. There could be several different pile-up algorithms and they will use a pile-up service which will be provided in the framework. The pile-up service will include all the common functionalities needed by the different pile-up algorithms. The current implementation is very dependent on SICB and does not use any pile-up service

The pile-up algorithm is responsible to manipulate the data from one or more data sources. It could, for example, modify the time of flight, to handle spill-over, or to include a label in one of the event data to know from which pile-up members they come.

The following job options are necessary to instantiate the current implementation of the pile-up structure. First, tell the `ApplicationMgr` to create a second `SicbEventSelector`, called `PileUpSelector`::

```
ApplicationMgr.ExtSvc = { "SicbEventCnvSvc","SicbEventSelector/EventSelector",
                          "SicbEventSelector/PileUpSelector" };
```

The application manager should know that the pile-up algorithm should be run, and the user has to configure how this algorithm works. That configuration depends on the concrete

implementation of the algorithm. As mentioned before, in the current implementation the only possibility is to extract the number of pile-up events as a function of the luminosity.

```
ApplicationMgr.TopAlg = { "PileUpAlg","Alg1",”Alg2”,...};
PileUpAlg.PileUpMode = "LUMI";
```

Finally the second event selector should be configured as the first one is in a normal job. The second input data source has to be defined by the PileUpSelector.JobInput property which accepts JOBID’s, FILE names or TAPEs:

```
PileUpSelector.JobInput = "JOBID 12933";
```


Chapter 15
Analysis utilities

15.1 Overview

In this chapter we give pointers to some of the third party software libraries that we use within Gaudi or recommend for use by algorithms implemented in Gaudi.

15.2 LHC++

The LHC++ project aims to replace the CERNLIB software libraries with a suite of OO software with roughly equivalent functionality. It consists of a number of distinct packages, both commercial and HEP specific. A complete list of LHC++ libraries (and their documentation) is maintained on the WWW at *http://wwwinfo.cern.ch/asd/lhc++/index.html*

The LHCb contact people for matters concerning LHC++ are Pavel Binko and Marco Cattaneo. You can also obtain help (and report problems) directly from the LHC++ team, preferably via their problem tracking system:
http://gnats.cern.ch/cgi-bin/wwwgnats.pl/LHCXX/1/1/

The following sections introduce some LHC++ components that are used (or can be used) in Gaudi.

15.3 CLHEP

CLHEP (“Class Library for High Energy Physics”) is a set of HEP-specific foundation and utility classes such as random generators, physics vectors, geometry and linear algebra. It is structured in a set of packages independent of any external package. The documentation for CLHEP can be found on WWW at *http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html*

CLHEP is used extensively inside Gaudi, in particular in the LHCbEvent and SicbCnv packages.

15.4 NAG C

The NAG C library is a commercial mathematical library providing a similar functionality to the FORTRAN mathlib (part of CERNLIB). It is organised into chapters, each chapter devoted to a branch of numerical or statistical computation. A full list of the functions is available at *http://wwwinfo.cern.ch/asd/lhc++/Nag_C/html/doc.html*

NAG C is not explicitly used in the Gaudi framework, but developers are encouraged to use it for mathematical computations. Instructions for linking NAG C with Gaudi can be found at *http://lhcb.cern.ch/~cattanem/LHCb/nagC.html*

Some NAG C functions print error messages to stdout by default, without any information about the calling algorithm and without filtering on severity level. A facility is provided by Gaudi to redirect these messages to the Gaudi MessageSvc. This is documented at *http://lhcb.cern.ch/~cattanem/LHCb/Gaudi/GaudiNagC.html*

Chapter 16
Visualization Facilities

16.1 Overview

In this chapter we describe how visualization facilities are provided to the applications based on the Gaudi framework. We present how we interface the physics event data objects, detector components or statistical objects to their graphical representation. One example of an application that uses the visualization services is an event display program. With this program we display graphically the event data from files or being acquired by the data acquisition. Another example could be an interactive analysis program that combines in the same application histogramming or manipulation of statistical entities, event display, and full interactive control by the end user of the data objects and algorithms of the application.

In the current release, these visualization services are at the level of a prototype. We have implemented the mechanism of converting event and detector objects into their graphical representation and built one example application. This application can serve as a proof of concept and can also be used to help in the development of physics algorithms (e.g. pattern recognition) or in the verification of the detector geometry.

16.1.1 The data visualization model

The Gaudi architecture envisaged implementing data visualization using a similar pattern to data persistency. We do not want to implement visualization methods in each *data object*. In other words, we do not want to tell an object to “draw” itself. Instead we would implement *converters* as separate entities that are able to create specific graphical representations for each type of data object and for each graphical package that we would like to use. In that way, as for the persistency case, we decouple the definition and behaviour of the data objects from the various technologies for graphics. We could configure at run time to have 2D or 3D graphics depending on the needs of the end-user at that moment.

Figure 30 illustrates the components that need to be included in an application to make it capable of visualizing data objects. The interactive user interface is a *Service* which allows the

end-user to interact with all the components of the application. The user could select which objects to display, which algorithms to run, what properties of which algorithm to inspect and modify, etc. This interaction can be implemented using a graphical user interface or by using a scripting language.

The User interface service is also in charge of managing one or more GUI windows where views of the graphical representations are going to be displayed.

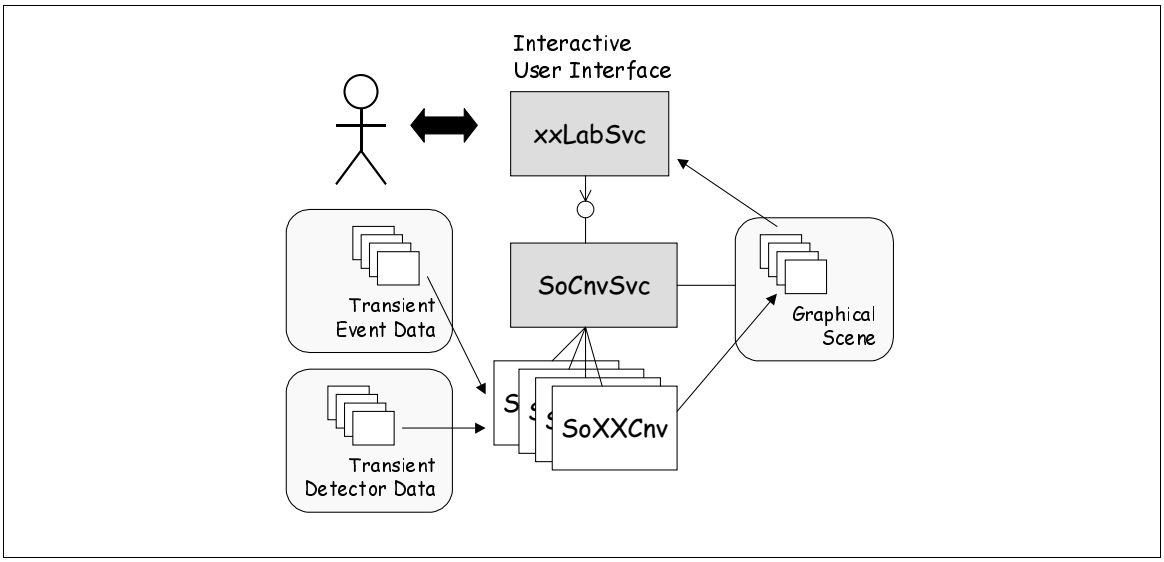


Figure 30 Components for visualization

The other main component is a *Conversion Service* that handles the conversion of objects into their graphical representation. This service requires the help of a number of specialized converters, one for each type of data object that needs to be graphically displayed. The transient store of graphical representations is shared by the conversion service, together with the converters, and the user interface component. The form of this transient store depends on the choice of graphics package. Typically it is the user interface component that would trigger the conversion service to start the conversion of a number of objects (next event), but this service can also be triggered by any algorithm that would like to display some objects.

16.2 Using the GaudiLab services

The current implementation of the user interface service is based on the Open Scientist suit of packages. Some documentation of these packages can be found in *http://www.lal.in2p3.fr/OpenScientist*. The following is the list we are using for implementing the `xmLabSvc`.

- **OpenGL**: 3D graphics
- **SoFree**: a free implementation of Open Inventor which runs on top of OpenGL
- **HEPVis**: a free collaborative set of classes for HEP over Open Inventor
- **Tcl**: a scripting language

- **Lab**: a package tieding all the others together to present a coherent environment

The Lab package proposes various graphical user interface to handle an interactive session. Under UNIX we use xmSession that is a direct Motif user interface. On NT it is planned to use the native windowing system, the win32Session (not yet available in the current release).

16.2.1 Interacting with it

3D examiner viewer controls are :

- right mouse button : popup menu.
- right / popup / decorations : enable the Inventor "decorations".
- Click in the "hand" : pass in viewing mode.
- Click in the "eye" : map the scene to the window size.
- Click in the "target" : center the scene over a picked point.
- left + ptr move : permits to rotate the scene.
- left + middle + ptr move : permit to scale the scene.
- ctrl + shift + left + ptr move : permits to scale the scene.
- ctrl + left + ptr move : permits to pane the scene.
- middle + ptr move : permit to pane the scene.

16.2.2 Writing graphic converters

The role of each converter So<xxx>Cnv is to produce an Open Inventor node that represents the object. The following fragment of code shows how this is done for the geometry of a detector element.The code has been simplyfied to be more illustrative. The 3D graphical objects that are created are standard OpenInventor objects (in bold).

Listing 56 Fragment of SoDetectorElementCnv

```
28:  StatusCode SoDetElemCnv::createRep(DataObject* aObject,IOpaqueAddress&*)
29:  {
30:    DetectorElement* de  = dynamic_cast<DetectorElement*>(aObject);
31:    ILVolume*    lv  = de->geometry()->lvolume();
32:    SolidBox*    box = dynamic_cast<SolidBox*>(lv->solid()->coverTop());
33:
34:    SoSeparator* separator     = new SoSeparator;
35:    SoDrawStyle* drawStyle     = new SoDrawStyle;
36:    SoMaterial*  material      = new SoMaterial;
37:    separator->addChild(drawStyle);
38:    separator->addChild(material);
39:    // set drawing styles
40:    drawStyle->style.setValue(SoDrawStyle::LINES);
41:    drawStyle->linePattern.setValue(0xFFFF);
42:    material->diffuseColor.setValue(SbColor(0.,1.,0.));
43:
44:    // Code related to the transformation
45:    SoTransform* trans = new SoTransform;
46:    ...
47:    separator->addChild(trans);
48:
49:    SoCube* cube = new SoCube();
50:    cube->width  = box->xHalfLength() * 2;
51:    cube->height = box->yHalfLength() * 2;
52:    cube->depth  = box->zHalfLength() * 2;
53:
54:    separator->addChild(cube);
55:    m_pSo->addNode(separator);
56:    return StatusCode::SUCCESS;
57:  }
```


Chapter 17
Design considerations

17.1 Generalities

In this chapter we look at how you might actually go about designing and implementing a real physics algorithm. It includes points covering various aspects of software development process and in particular:

- The need for more “thinking before coding” when using an OO language like C++.
- Emphasis on the specification and analysis of an algorithm in mathematical and natural language, rather than trying to force it into (unnatural?) object orientated thinking.
- The use of OO in the design phase, i.e. how to map the concepts identified in the analysis phase into data objects and algorithm objects.
- The identification of classes which are of general use. These could be implemented by the computing group, thus saving you work!
- The structuring of your code by defining private utility methods within concrete classes.

When designing and implementing your code we suggest the your priorities should be as follows: (1) Correctness, (2) Clarity, (3) Efficiency and, very low in the scale, OOness

Tips about specific use of the C++ language can be found in the coding rules document [4] or specialized literature.

17.2 Designing within the Framework

A physicist designing a real physics algorithm does not start with a white sheet of paper. The fact that he or she is using a framework imposes some constraints on the possible or allowed

designs. The framework defines some of the basic components of an application and their interfaces and therefore it also specifies the places where concrete physics algorithms and concrete data types will fit in with the rest of the program. The consequences of this are: on one hand, that the physicists designing the algorithms do not have complete freedom in the way algorithms may be implemented; but on the other hand, neither do they need worry about some of the basic functionalities, such as getting end-user options, reporting messages, accessing event and detector data independently of the underlying storage technology, etc. In other words, the framework imposes some constraints in terms of interfaces to basic services, and the interfaces the algorithm itself is implementing towards the rest of the application. The definition of these interfaces establishes the so called “master walls” of the data processing application in which the concrete physics code will be deployed. Besides some general services provided by the framework, this approach also guarantees that later integration will be possible of many small algorithms into a much larger program, for example the LHCb reconstruction program. In any case, there is still a lot of room for design creativity when developing physics code within the framework and this is what we want to illustrate in the next sections.

To design a physics algorithm within the framework you need to know very clearly what it should do (the requirements). In particular you need to know the following:

- What is the input data to the algorithm? What is the relationship of these data to other data (e.g. event or detector data)?
- What new data is going to be produced by the algorithm?
- What’s the purpose of the algorithm and how is it going function? Document this in terms of mathematical expressions and plain english.[1]
- What does the algorithm need in terms of configuration parameters?
- How can the algorithm be partitioned (structured) into smaller “algorithm chunks” that make it easier to develop (design, code, test) and maintain?
- What data is passed between the different chunks? How do they communicate?
- How do these chunks collaborate together to produce the desired final behaviour? Is there a controlling object? Are they self-organizing? Are they triggered by the existence of some data?
- How is the execution of the algorithm and its performance monitored (messages, histograms, etc.)?
- Who takes the responsibility of bootstrapping the various algorithm chunks.

For didactic purposes we would like to illustrate some of these design considerations using a hypothetical example. Imagine that we would like to design a tracking algorithm for LHCb based on a Kalman-filter algorithm.

1. Catalan is also acceptable.

17.3 Analysis Phase

As mentioned before we need to understand in detail what the algorithm is supposed to do before we start designing it and of course before we start producing lines of C++ code. One old technique for that, is to think in terms of data flow diagrams, as illustrated in Figure 31, where we have tried to decompose the tracking algorithm into various processes or steps.

In the analysis phase we identify the data which is needed as input (event data, geometry data, configuration parameters, etc.) and the data which is produced as output. We also need to think about the intermediate data. Perhaps this data may need to be saved in the persistency store to allow us to run a part of the algorithm without starting always from the beginning.

We need to understand precisely what each of the steps of the algorithm is supposed to do. In case a step becomes too complex we need to sub-divide it into several ones. Writing in plain english and using mathematics whenever possible is extremely useful. The more we understand about what the algorithm has to do the better we are prepared to implement it.

17.4 Design Phase

We now need to decompose our physics algorithm into one or more *Algorithms* (as framework components) and define the way in which they will collaborate. After that we need to specify the data types which will be needed by the various *Algorithms* and their relationships. Then, we need to understand if these new data types will be required to be stored in the persistency store and how they will map to the existing possibilities given by the object persistency technology. This is done by designing the appropriate set of *Converters*. Finally, we need to identify utility classes which will help to implement the various algorithm chunks.

17.4.1 Defining Algorithms

Most of the steps of the algorithm have been identified in the analysis phase. We need at this moment to see if those steps can be realized as framework *Algorithms*. Remember that an *Algorithm* from the view point of the framework is basically a quite simple interface (initialize, execute, finalize) with a few facilities to access the basic services. In the case of our hypothetical algorithm we could decide to have a “master” *Algorithm* which will orchestrate the work of a number of *sub-Algorithms*. This master *Algorithm* will be also be in charge of bootstraping them. Then, we could have an *Algorithm* in charge of finding the tracking seeds, plus a set of others, each one associated to a different tracking station in charge of propagating a proto-track to the next station and deciding whether the proto-track needs to be kept or not. Finally, we could introduce another Algorithm in charge of producing the final tracks from the surviving proto-tracks.

It is interesting perhaps in this type of algorithm to distribute parts of the calculations (extrapolations, etc.) to more sophisticated “hits” than just the unintelligent original ones. This could be done by instantiating new data types (clever hits) for each event having

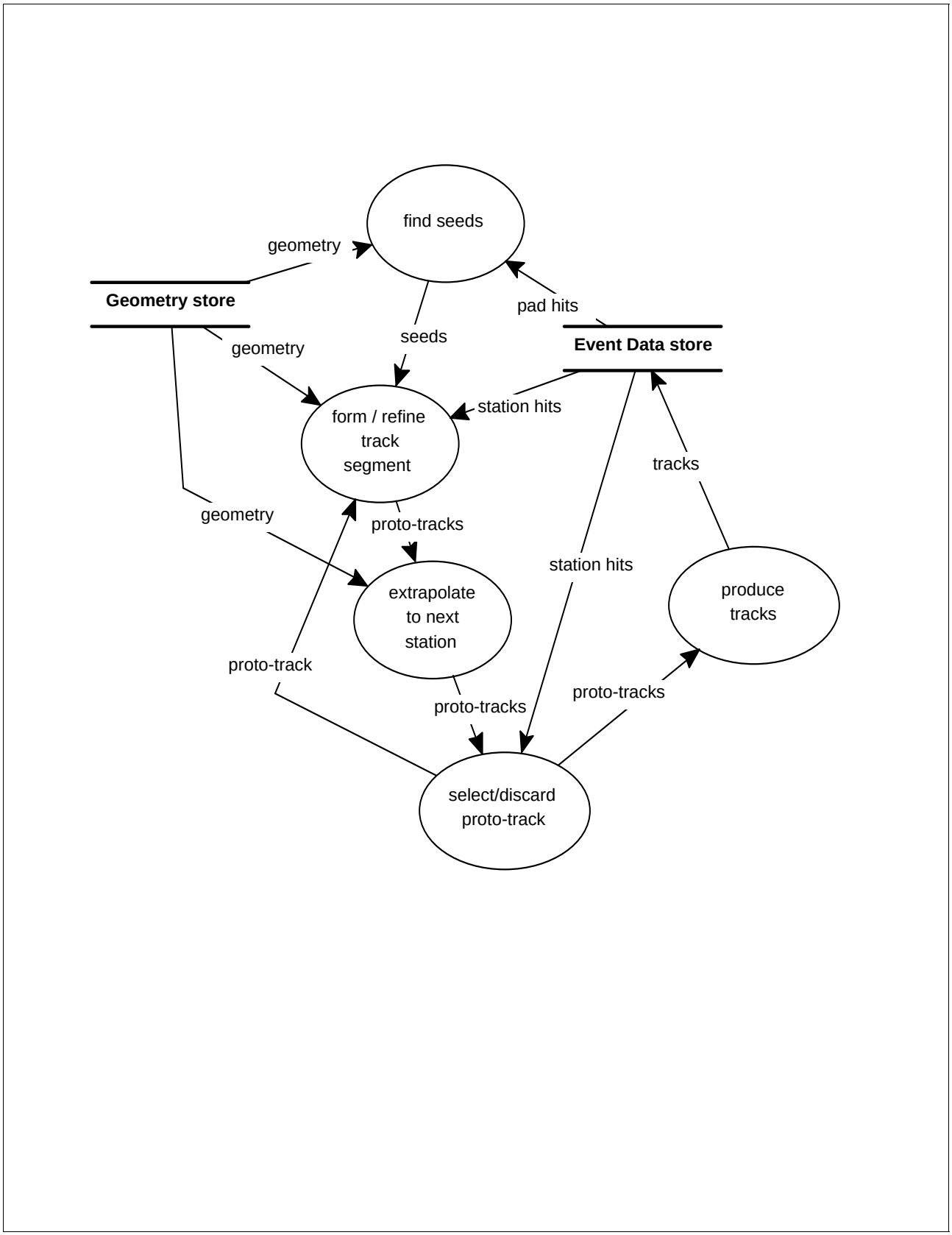


Figure 31 Hypothetical decomposition of a tracking algorithm based on a Kalman filter using a Data flow Diagram

references to the original hits. For that, it would be required to have another Algorithm whose role is to prepare these new data objects, see Figure 32.

The master *Algorithm* (TrackingAlg) is in charge of setting up the other algorithms and scheduling their execution. It is the only one that has a global view but it does not need to know the details of how the different parts of the algorithm have been implemented. The application manager of the framework only interacts with the master algorithm and does not need to know that in fact the tracking algorithm is implemented by a collaboration of Algorithms.

17.4.2 Defining Data Objects

The input, output and intermediate data objects need to be specified. Typically, the input and output are specified in a more general way (algorithm independent) and basically are pure data objects. This is because they can be used by a range of different algorithms. We could have various types of tracking algorithm all using the same data as input and producing similar data as output. On the contrary, the intermediate data types can be designed to be very algorithm dependent.

The way we have chosen to communicate between the different *Algorithms* which constitute our physics algorithm is by using the transient event data store. This allows us to have low coupling between them, but other ways could be easily envisaged. For instance, we could implement specific methods in the algorithms and allow other “friend” algorithms to use them directly.

Concerning the relationships between data objects, it is strongly discouraged to have links from the input data objects to the newly produced ones (i.e. links from hits to tracks). In the other direction this should not be a problem (i.e from tracks to constituent hits).

For data types that we would like to save permanently we need to implement a specific *Converter*. One converter is required for each type of data and each kind of persistency technology that we wish to use. This is not the case for the data types that are used as intermediate data, since this data is completely transient.

17.4.3 Mathematics and other utilities

It is clear that to implement any algorithm we will need the help of a series of utility classes. Some of these classes are very generic and they can be found in common class libraries. For example the standard template library. Other utilities will be more high energy physics specific, especially in cases like fitting, error treatment, etc. We envisage making as much use of these kinds of utility classes as possible.

Some algorithms or algorithm-parts could be designed in a way that allows them to be reused in other similar physics algorithms. For example, perhaps fitting or clustering algorithms could be designed in a generic way such that they can be used in various concrete algorithms. During design is the moment to identify this kind of re-usable component or to identify existing ones that could be used instead and adapt the design to make possible their usage.

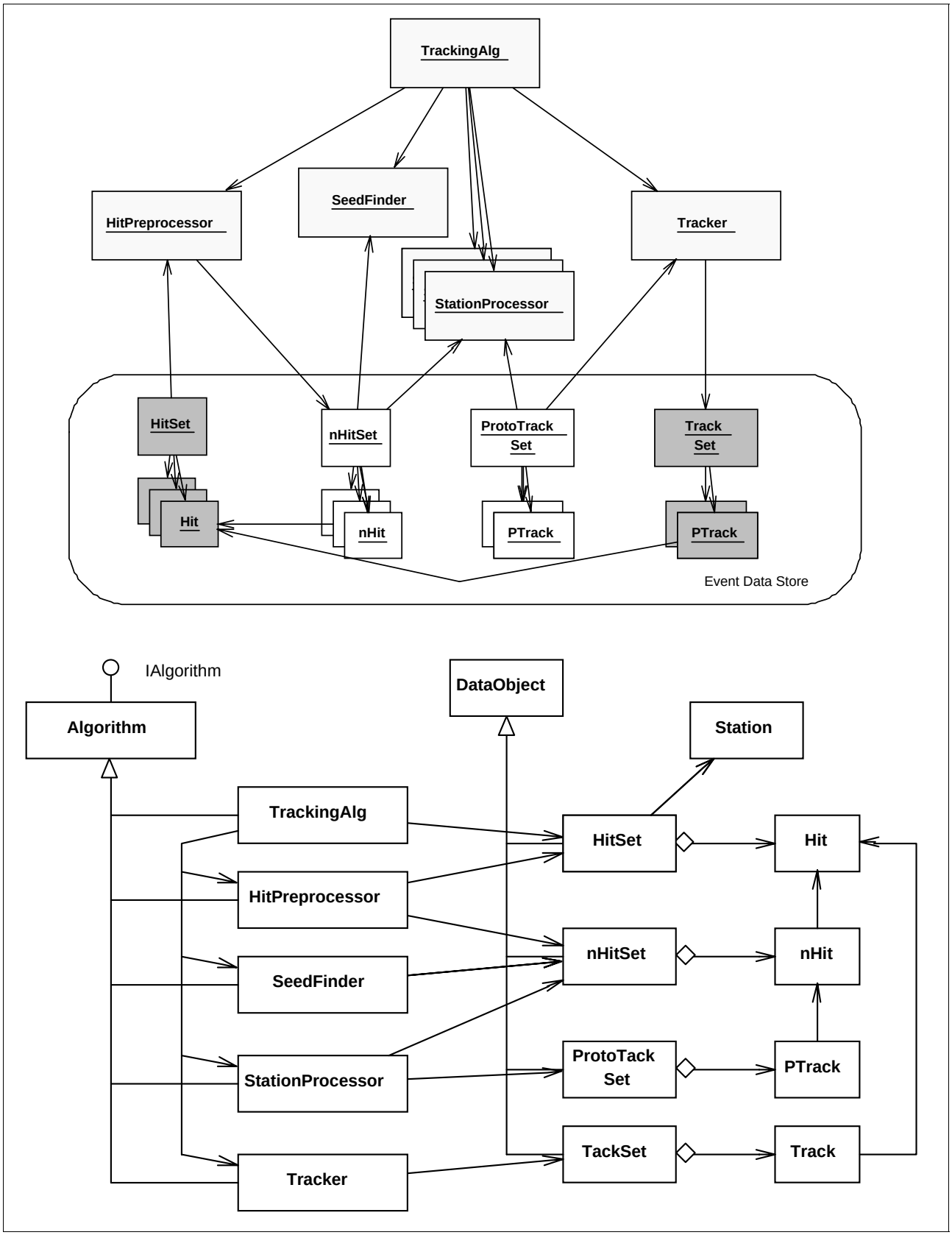


Figure 32 Object diagram (a) and class diagram (b) showing how the complete example tracking algorithm could be decomposed into a set of specific algorithms that collaborate to perform the complete task.

Appendix A
References

1 GAUDI - Architecture Design Report [LHCb 98-064 COMP]

2 GAUDI online code documentation
(http://lhcb.cern.ch/LHCbSoft/RefManual/v3/LHCbSoft.html)

3 GAUDI - User Requirements Document [LHCb 98-065 COMP]

4 LHCb coding conventions [LHCb 98-049 COMP]

Appendix B
Options for standard components

The following is a list of options that may be set for the standard components: e.g. data files for input, print-out level for the message service, etc. The options are listed in tabular form for each component along with the default value and a short explanation. The component name is given in the table caption thus: [ComponentName].

Table B.1 Standard Options for the Application manager [ApplicationMgr]

Option name	Default value	Meaning
EvtSel	“FILE S:/datafile.dat”;	Input event data. Format: “FILE <filename1>[,<filename2>,...]”; “JOBID <id1>[, <id2>,...]” “TAPE <vsn1-file1>[, <vsn2-file2>,...]" "NONE" (if no event input)
~~EvtMax~~[a]	~~-1;~~	~~Maximum number of events to process.~~
TopAlg		List of top level algorithms. Format: {<Type>/<Name>[, <Type2>/<Name2>,...]};
ExtSvc		List of external services names (not known to the ApplicationMgr). Format: {<Type>/<Name>[, <Type2>/<Name2>,...]};
OutStream		Declares an output stream object for writing data to a persistent store, e.g. {“DstWriter”}; See also Table B.6
Dlls		Search list of DLLs for dynamic loading (NT only). Format: {<dll1>[,<dll2>,...]};
DetStorageType	0;	Detector database storage type 7=XML
DetDbLocation	“empty”;	Detector database location (filename,URL)
DetDbRootName	"empty";	Name of the root node of the detector transient store

Table B.1 Standard Options for the Application manager [ApplicationMgr]

Option name	Default value	Meaning
The last two options define the source of the job options file and so they cannot be defined in the job options file itself. There are two possibilities to set these options, the first one is using a environment variable called JOBOPTPATH or setting the option to the application manager directly from the main program[b]. The coded option takes precedence.		
JobOptionsType	“FILE”;	Type of file (FILE implies ascii)
JobOptionsPath	“jobOptions.txt”;	Path for job options source

a. This option is obsolete and is only for backward compatibility. Use [EventSelector].EvtMax instead

b. The setting of properties from the main program is discussed in chapter 4.

Table B.2 Standard Options for the message service [MessageSvc]

Option name	Default value	Meaning
OutputLevel	0;	Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL
Format	“% F%18W%S%7W%R%T %0W%M”;	Format string.

Table B.3 Standard Options for all algorithms [<myAlgorithm>]

Any algorithm derived from the Algorithm base class can override the global Algorithm options thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0;	Message Service Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO,4=WARNING, 5=ERROR, 6=FATAL
Enable	true	If false, application manager skips execution of this algorithm
ErrorMax	1	Job stops when this number of errors is reached
ErrorCount	0	Current error count
ProfileInitialize	false;	Enable / Disable profiling of Algorithm initialisation
ProfileExecute	true;	Enable / Disable profiling of Algorithm execution
ProfileFinalize	false;	Enable / Disable profiling of Algorithm finalisation

Table B.4 Standard Options for all services [<myService>]

Any service derived from the Service base class can override the global `MessageSvc.OutputLevel` thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0;	Message Service Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO,4=WARNING, 5=ERROR, 6=FATAL

Table B.5 Standard Options for all Tools [<myTool>]

Any tool derived from the AlgTool base class can override the global `MessageSvc.OutputLevel` thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0;	Message Service Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO,4=WARNING, 5=ERROR, 6=FATAL

Table B.6 Options available for output streams (e.g. DstWriter)

Output stream objects are used for writing user created data into data files or databases. They are created and named by setting the option `ApplicationMgr.OutStream`. For each output stream the following options are available		
Option name	**Default value**	**Meaning**
ItemList		The list of data objects to be written to this stream, e.g. {“/Event#1”,”Event/MyTracks/#1”};
EvtDataSvc	“EventDataSvc”;	The service from which to retrieve objects.
EvtConversionSvc	“EventConversionSvc”;	Conversion service to be used.
OutputFile	“”;	Output file name

Table B.7 Standard Options for the event persistency service [PersistencySvc]

Option name	**Default value**	**Meaning**
CnvServices	{““};	Sets up the event persistency service. Possible values are: {“RootEventCnvSvc”}; {“SicbEventCnvSvc”}; or both together

Table B.8 Standard Options for the histogram service [HistogramPersistencySvc]

Option name	**Default value**	**Meaning**
OutputFile		Output file for histograms. No output if not defined

Table B.9 Standard Options for the N-tuple service [NTupleSvc]

Option name	Default value	Meaning
Input	{“”};	Input file(s) for n-tuples. Format: {“<RZ directory1>#<filename1>” [,“<RZ directory2>#<filename2>”,...]}
Output	{“”};	Output file for n-tuples. Format: {“<RZ directory1>#<filename1>” [,“<RZ directory2>#<filename2>”,...]}
Type	6;	Storage type. 6=HBOOK

Table B.10 Standard Options for the Sicb event selector [EventSelector]

Option name	Default value	Meaning
~~TapesDatabase~~[a]	~~{“$LHCBHOME/mcdbase/mcmain.db”, “$LHCBHOME/mcdbase/mcupdate.db”};~~	~~Tapes database file location~~
FirstEvent	1;	First event to process (allows skipping of preceding events)
EvtMax	-1;	Maximum number of events to process
PrintFreq	10;	Frequency with which event number is reported
JobInput	““;	String of input files (same format as ApplicationMgr.EvtSel), used only for pileup event selector(s)

a. This option is obsolete, it is provided only for backwards compatibility and will be removed in a future release. Since release 5, the framework accesses the ORACLE book-keeping database via the oraweb01.cern.ch web server.

Table B.11 Standard Options for Sicb Pileup Algorithm [PileUpAlg]

Option name	Default value	Meaning
PileUpMode	““;	Pileup mode. Only “LUMI”; is implemented

Table B.12 Standard Options for Particle Properties Service [ParticlePropertiesSvc]

Option name	Default value	Meaning
ParticlePropertiesFile	“($LHCBDBASE)/cdf/particle.cdf”;	Particle properties database location

Table B.13 Standard Options for Random Numbers Generator Service [RndmGenSvc]

Option name	Default value	Meaning
Engine	“HepRndm::Engine<RanluxEngine>”;	Random number generator engine
Seeds		Table of generator seeds
Column	0;	Number of columns in seed table -1
Row	1;	Number of rows in seed table -1
Luxury	3;	Luxury value for the generator
UseTable	false;	Switch to use seeds table

Table B.14 Standard Options for Transport Service [TransportSvc]

Option name	Default value	Meaning
ChronoStatService	“ChronoStatSvc”;	Name of the Chrono Service to be used
MagneticFieldService	“MagneticFieldSvc”;	Name of the Magnetic Field Service to be used
DetectorDataService	“DetectorDataSvc”;	Name of the Detector Data Service to be used
StandardGeometryTop	“/dd/Structure/LHCb”;	Location of the top node of the detector geometry in the transient detector data store

Table B.15 Standard Options for Chrono and Stat Service [ChronoStatSvc]

Option name	Default value	Meaning
ChronoPrintOutTable	true;	Global switch for profiling printout
PrintUserTime	true;	Switch to print User Time
PrintSystemTime	false;	Switch to print System Time
PrintEllapsedTime	false	Switch to print Elapsed time (Note typo in option name!)
ChronoDestinationCout	false;	If true, printout goes to cout rather than MessageSvc
ChronoPrintLevel	3;	Print level for profiling (values as for MessageSvc)
ChronoTableToBeOrdered	true;	Switch to order printed table
StatPrintOutTable	true;	Global switch for statistics printout
StatDestinationCout	false;	If true, printout goes to cout rather than MessageSvc
StatPrintLevel	3;	Print level for profiling (values as for MessageSvc)
StatTableToBeOrdered	true;	Switch to order printed table

Table B.16 Standard Options for XML conversion service [XmlCnvSvc]

Option name	Default value	Meaning
AllowGenericConversion	false;	Switch for generic detector element conversion[a]

a. The XML conversion service allows the possibility to convert user defined detector elements to generic detector elements. This means that only the generic part of the detector element, and its associated geometry, will be converted but not the user defined detector element data. This feature can be used when the user defined detector element data are not needed (e.g. visualization) or when the corresponding user defined XML converters are not available (testing). When this feature is switched ON, the Gaudi application will run successfully with an information message saying that this feature is enabled, and will print out the information about all the detector elements to which this generic conversion is applied. The limitation of this feature is that after the generic conversion the returned reference points to a `DetectorElement` object and not to the user defined class. This means that `SmartDataPtr` class can be parameterized only by `DetectorElement` class and not by the user defined class.

Table 11 Options of Algorithms in GaudiAlg package

Algorithm name	Option Name	Default value	Meaning
EventCounter	Frequency	1;	Frequency with which number of events should be reported
Prescaler	PercentPass	100.0;	Percentage of events that should be passed
Sequencer	Members		Names of mambers of the sequence
Sequencer	StopOverride	false;	If true, do not stop sequence if a filter fails

Appendix C
Job Options Grammar and Error Codes

C.1 The EBNF grammar of the Job Options files

The syntax of the Job-Options-File is defined through the following EBNF-Grammar.

```
Job-Options-File =
   {Statements} .

Statements =
   {Include-Statement} | {Assign-Statement} | {Append-Statement} |
{Platform-Dependency} .

AssertableStatements =
   {Include-Statement} | {Assign-Statement} | {Append-Statement} .

AssertionStatement =
   '#ifdef' | '#ifndef' .

Platform-Dependency =
   AssertionStatement 'WIN32' <AsertableStatements> [ #else <Asserta-
bleStatements> ] #endif

Include-Statement =
   '#include' string .

Assign-Statement =
   Identifier '.' Identifier '=' value ';' .

Append-Statement =
   Identifier '.' Identifier '+=' value ';' .
```


```
Identifier =
   letter {letter | digit} .

value =
   boolean | integer | double | string | vector .

vector =
   ‘{’ vectorvalue { ‘,’ vectorvalue } ‘}’ .


vectorvalue =
   boolean | integer | double | string .

boolean =
   ‘true’ | ‘false’ .

integer =
   prefix scientificdigit .

double =
   ( prefix <digit> ‘.’ [ scientificdigit ] ) |
   ( prefix ‘.’ scientificdigit ) .

string =
   ‘”’ {char} ‘”’ .

scientificdigit =
   < digit> [ ( ‘e’ | ‘E’ ) < digit> ] .

digit =
   <figure> .

prefix =
   [ ‘+’ | ‘-’ ] .

figure =
   ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’.

char =
```

any character from the ASCII-Code

```
letter =
```

set of all capital- and non-capital letter

C.2 Job Options Error Codes and Error Messages

The table below lists the error codes and error messages that the Job Options compiler may generate, their reason and how to avoid them.

Table 12 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #000 Internal compiler error	-	This code normally should never appear. If this code is shown there is maybe a problem with your memory, your disk-space or the property-file is corrupted.
Error #001 Included property-file does not exists or can not be opened	* wrong path in #include-directive * wrong file or mistyped filename * file is exclusively locked by another application * no memory available to open this file	Please check if any of the listed reasons occured in your case.
Warning #001 File already included by another file	The file was already included by another file and will not be included a second time. The compiler will ignore this #include-directive and will continue with the next statement.	Remove the #include-directive
Error #002 syntax error: Object expected	The compiler expected an object at the given position.	Maybe you mistyped the name of the object or the object contains unknown characters or does not fit the given rules.
Error #003 syntax error: Missing dot between Object and Propertyname	The compiler expect a dot between the Object and the Propertyname.	Check if the dot between the Object and the Propertyname is missing.
Error #004 syntax error: Identifier expected	The compiler expected an identifier at the given position.	Maybe you mistyped the name of the identifier or the identifier contains unknown characters or does not fit the given rules.
Error #005 syntax error: Missing operator '+=' or '='	The compiler expected an operator between the Propertyname and the value.	Check if there is a valid operator after the Propertyname. Note that a blank or tab is not allowed between '+='!

Table 12 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #006 String is not terminated by a “	A string (value) was not terminated by a “.	Check if all your strings are beginning and ending with “. Note that the position given by the compiler can be wrong because the compiler may thought that following statements are part of the string!
Error #007 syntax error: #include-statement is not correct	The next token after the #include is not a string.	Make sure that after the #include-directive there is specified the file to include. The file must be defined as a string!
Error #008 syntax error: #include does not end with a ;	The include-directive was terminated by a ;	Remove the ; after the #include-directive.
Error #009 syntax error: Values must be separated with ’,’	One or more values within a vector were not separated with a ’,’ or one ore more values within a vector are mistyped.	Check if every value in the vector is separated by a ’,’. If so the reason for this message may result in mistyped values in the vector (maybe there is a blank or tab between numbers).
Error #010 syntax error: Vector must end with ’}’	The closing bracket is missing or the vector is not terminated correctly.	Check, if the vector ends with a ’}’ and if there is no semicolon before the ending-bracket.
Error #011 syntax error: Statement must end with a ;	The statement is not terminated correctly.	Check if the statement ends with a semicolon ’;’.
Runtime-Error #012: Cannot append to object because it does not exists	The compiler cannot append the values to the object.propertyname because the object does not exist.	Check if the refered object is defined in one of the included files, if so check if you writed the object-name exactly like in the include-file.
Runtime-Error #013 Cannot append to object because Property does not exists	The compiler cannot append the values to the object.propertyname because the property does not exist.	Check if there was already something assigned to the refered property (in the include-file or in the current file). If not then modify the append-statement into a assign-statement. If there was already something assigned, check if the object-name and the property-name are typed correctly.

Table 12 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #014 Elements in the vector are not of the same type	One or more elements in the vector have a different type than the first element in the vector. All elements must have the same type like the first declarated element.	Check declaration of vector, check the types and check, if maybe a value is mistyped.
Error #015 Value(s) expected	The compiler didn't find values to append or assign	Check the statement if there exists values and if they are written correctly. Maybe this error is a result of a previous error!
Error #016 Specified property-file does not exist or can not be resolved	The compiler was not able to include a property-file or didn't found the file. A reason can be that the compiler was not able to resolve an environment-variable which points to the location of the property-file.	Check if you are using enviornment-variables to resolve the file, if they are mistyped (wether in the system or in the #include-directive) or not set correctly.
Error #017 #ifdef not followed by an identifier	The #ifdef-statement is not followed by the assertion-identifier (WIN32).	Add WIN32 after the #ifdef-statement.
Error #018 identifier in #ifdef / #ifndef not known	The assertion-identifier used in the #ifdef- /#ifndef-statement is not known. At the moment there can only be used WIN32!	Change identifier to WIN32.
Error #019 #ifdef / #ifndef / #else / #endif doesn't end with a ';'	A semicolon was found after the #ifdef- / #ifndef- / #else- / #endif-statement. These statements don't end with a semicolon.	Remove the semicolon after the #ifdef / #ifndef / #else / #endif-statement.

Appendix D
LHCb Event Data Model

In this Appendix we present the UML diagrams relating the Transient Event Data model.

.

Top Level LHCb Event Structures

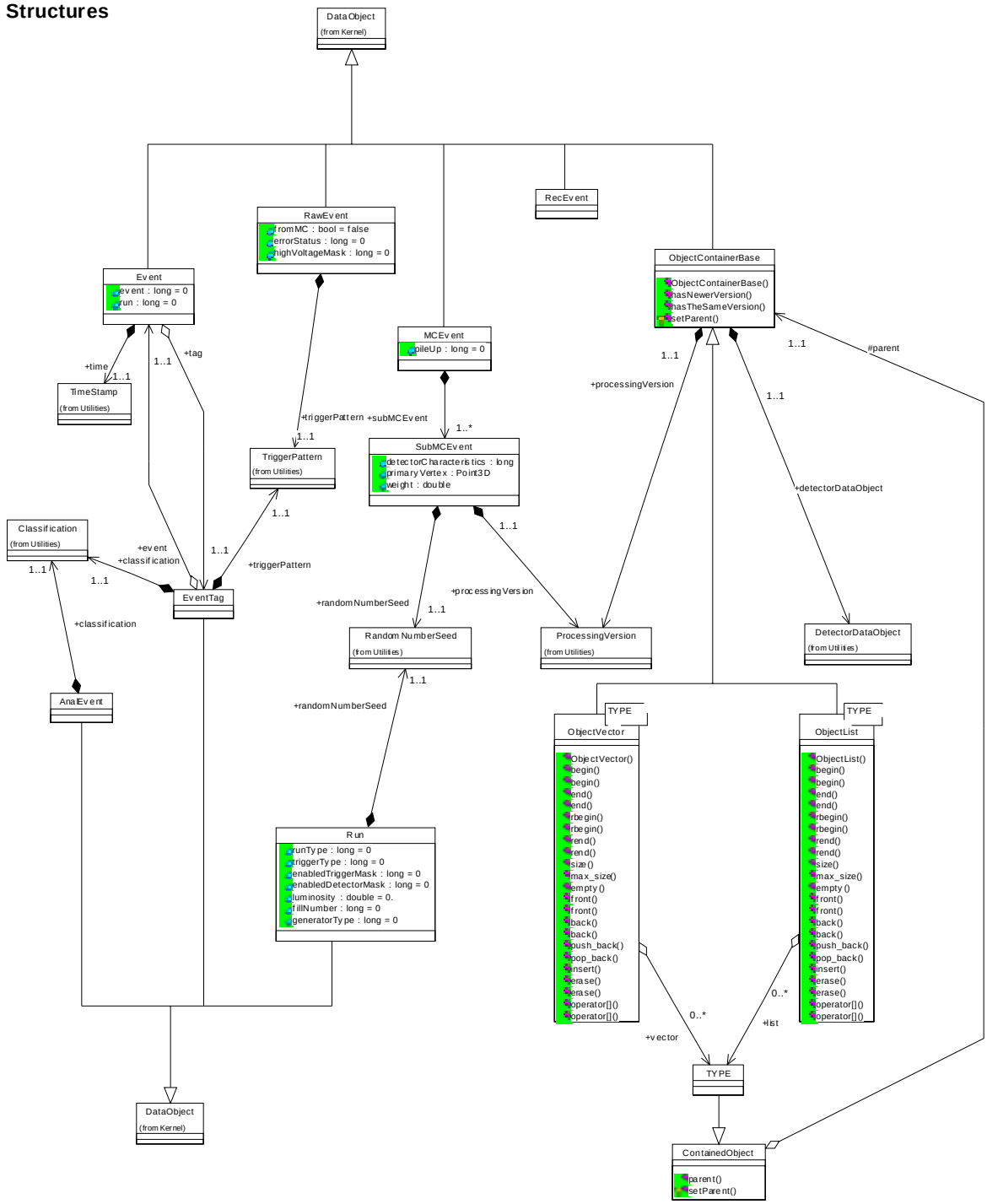


Figure D.1 Class diagram of the LHCb top level event classes.

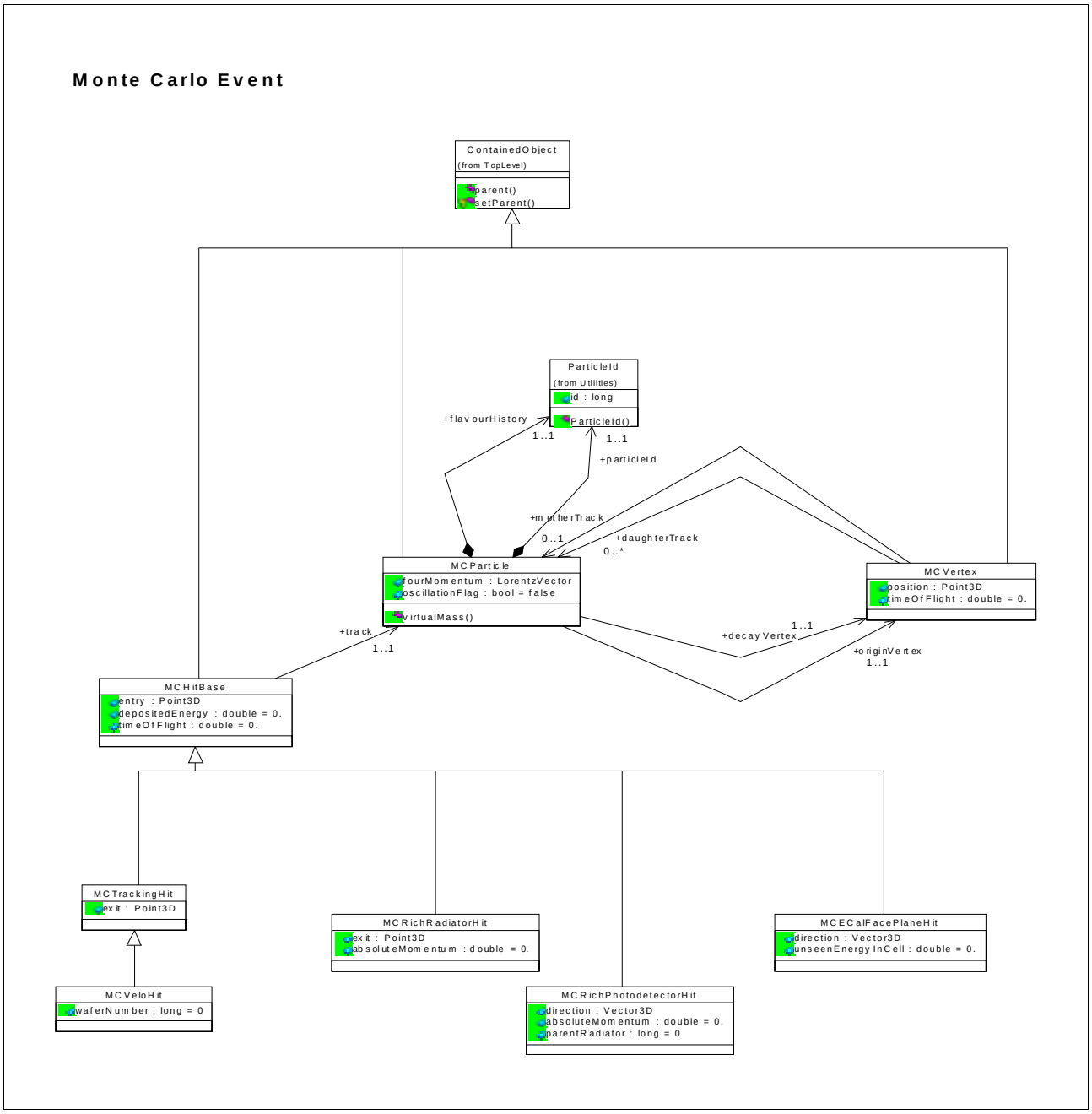


Figure D.2 Class definitions of Monte Carlo event.

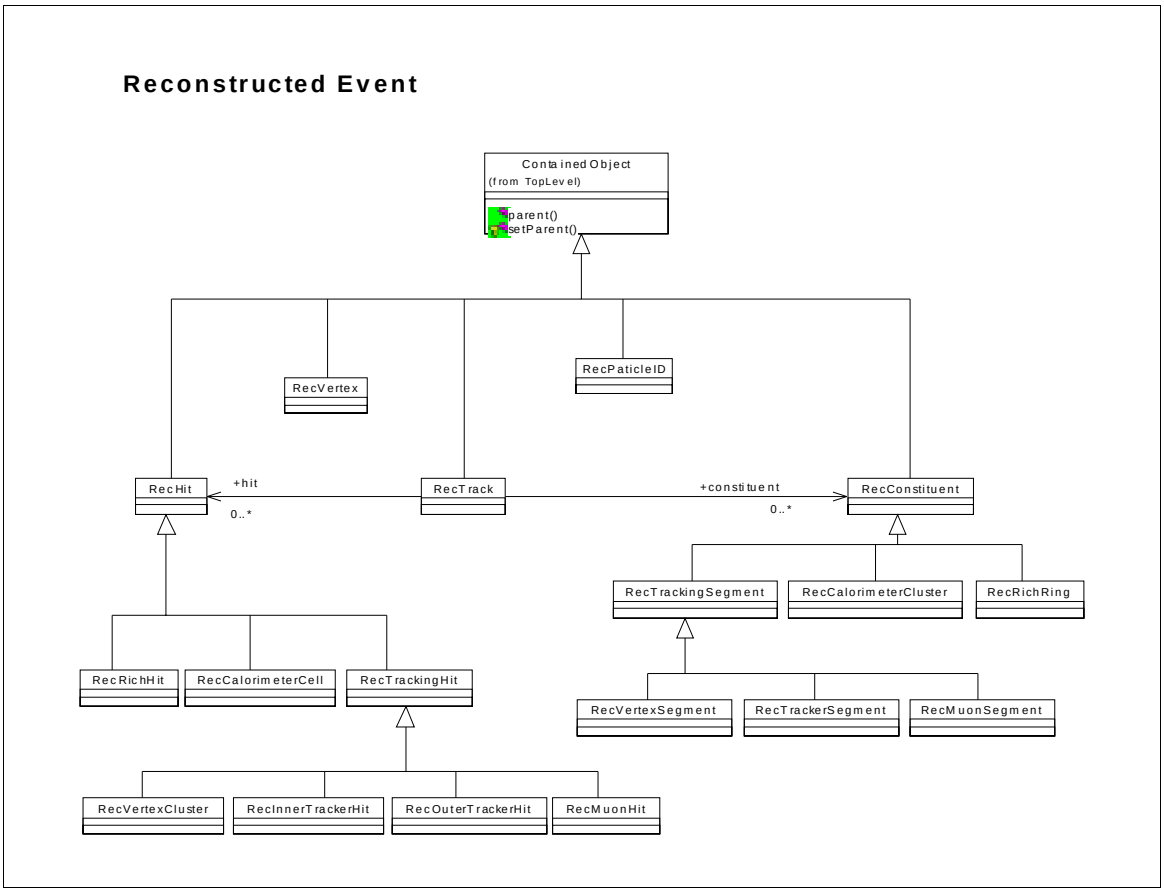


Figure 33 Data model of reconstructed event.

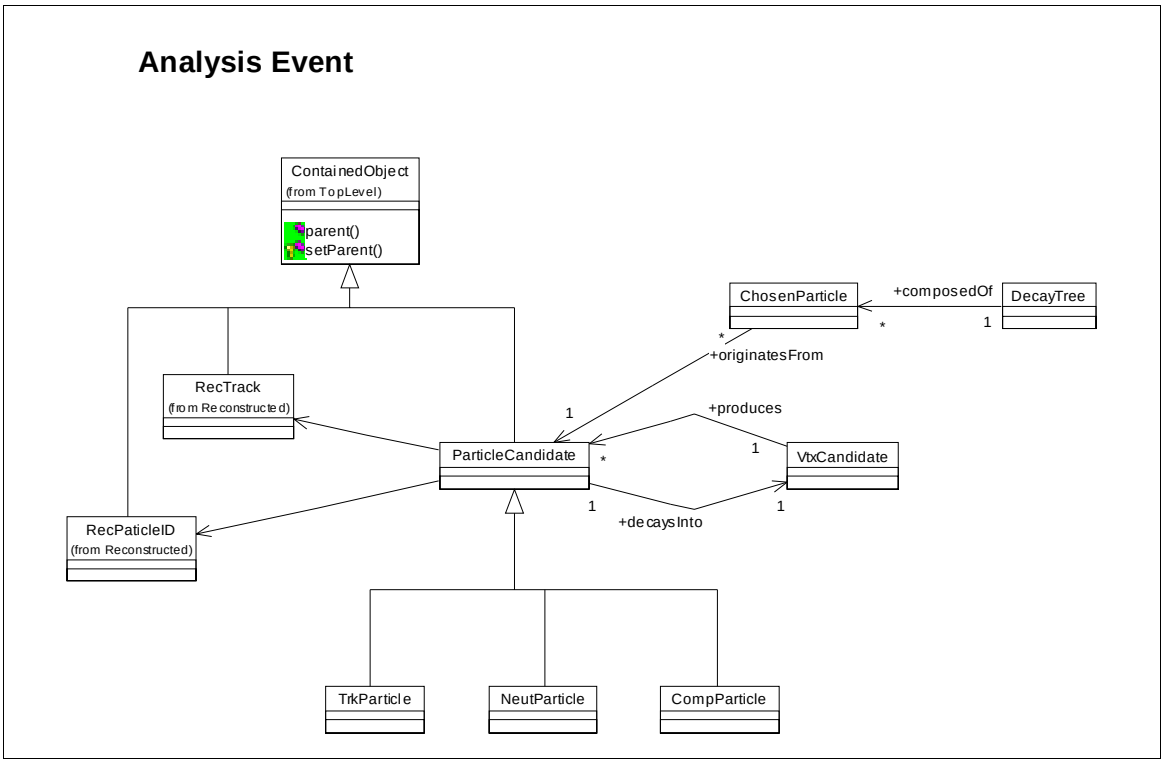


Figure 34 Data model of analysis event.

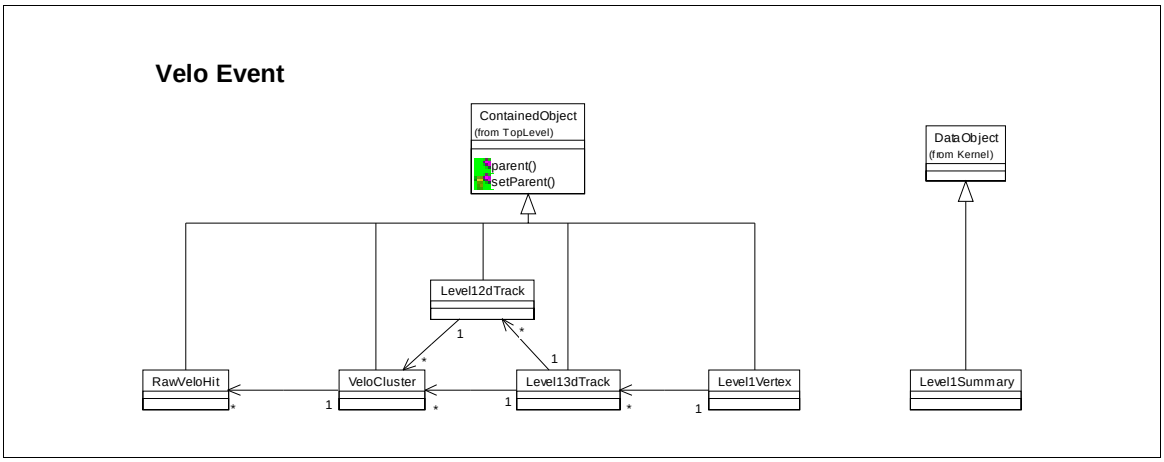


Figure 35 Velo event data model.

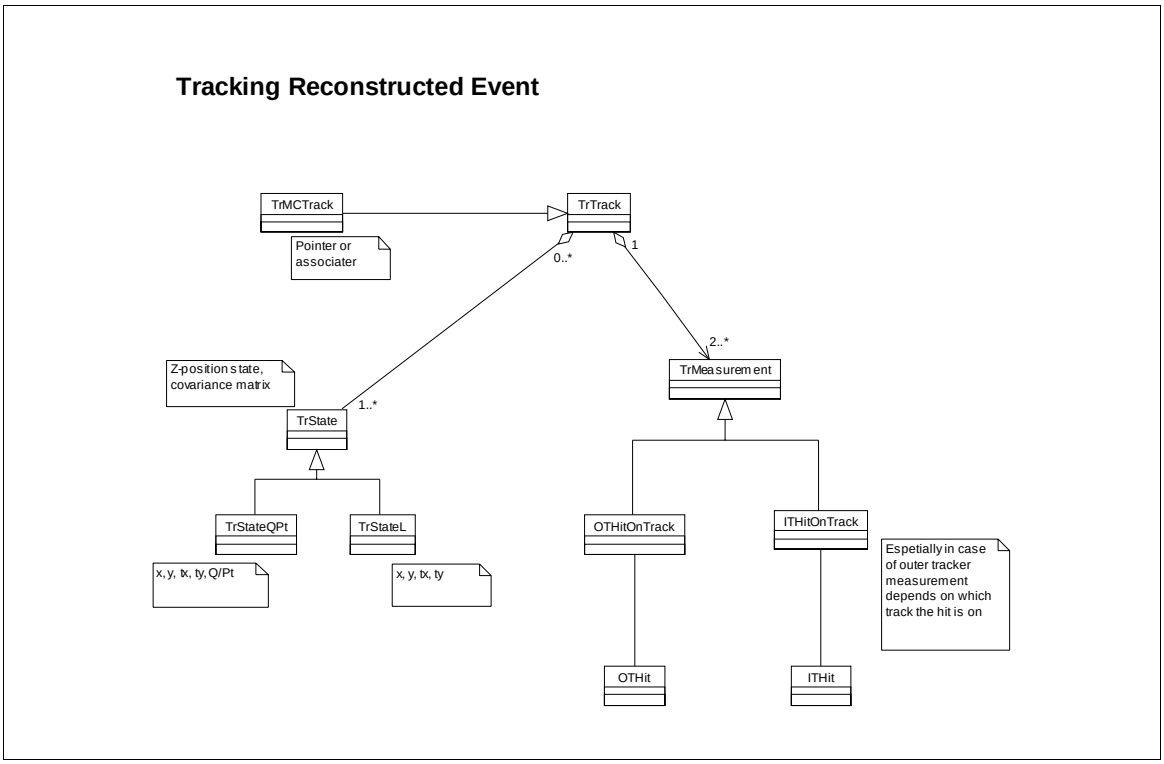


Figure 36 Tracking reconstructed event data model.

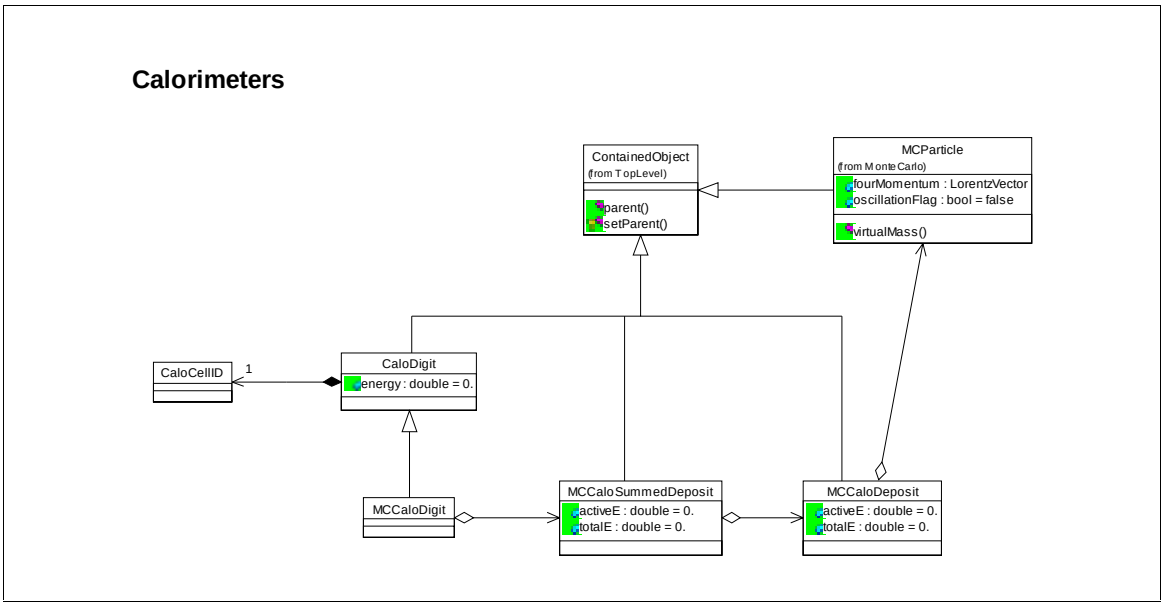


Figure 37 Event data model for Calorimeters.

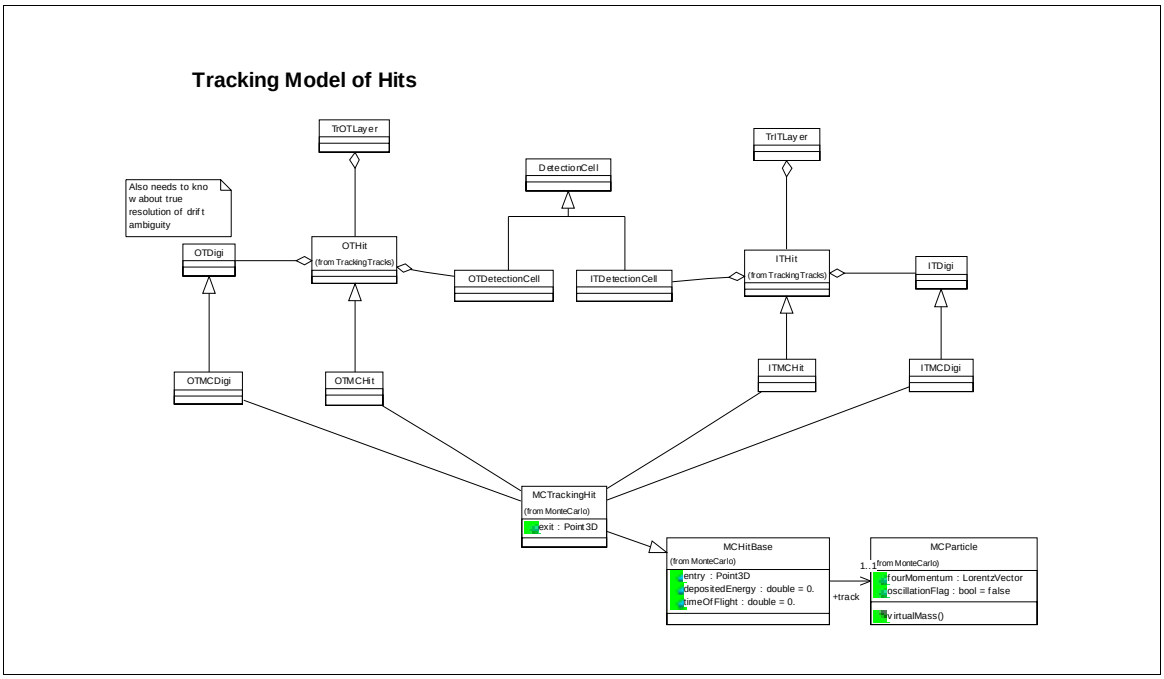


Figure 38 Tracking model of hits.

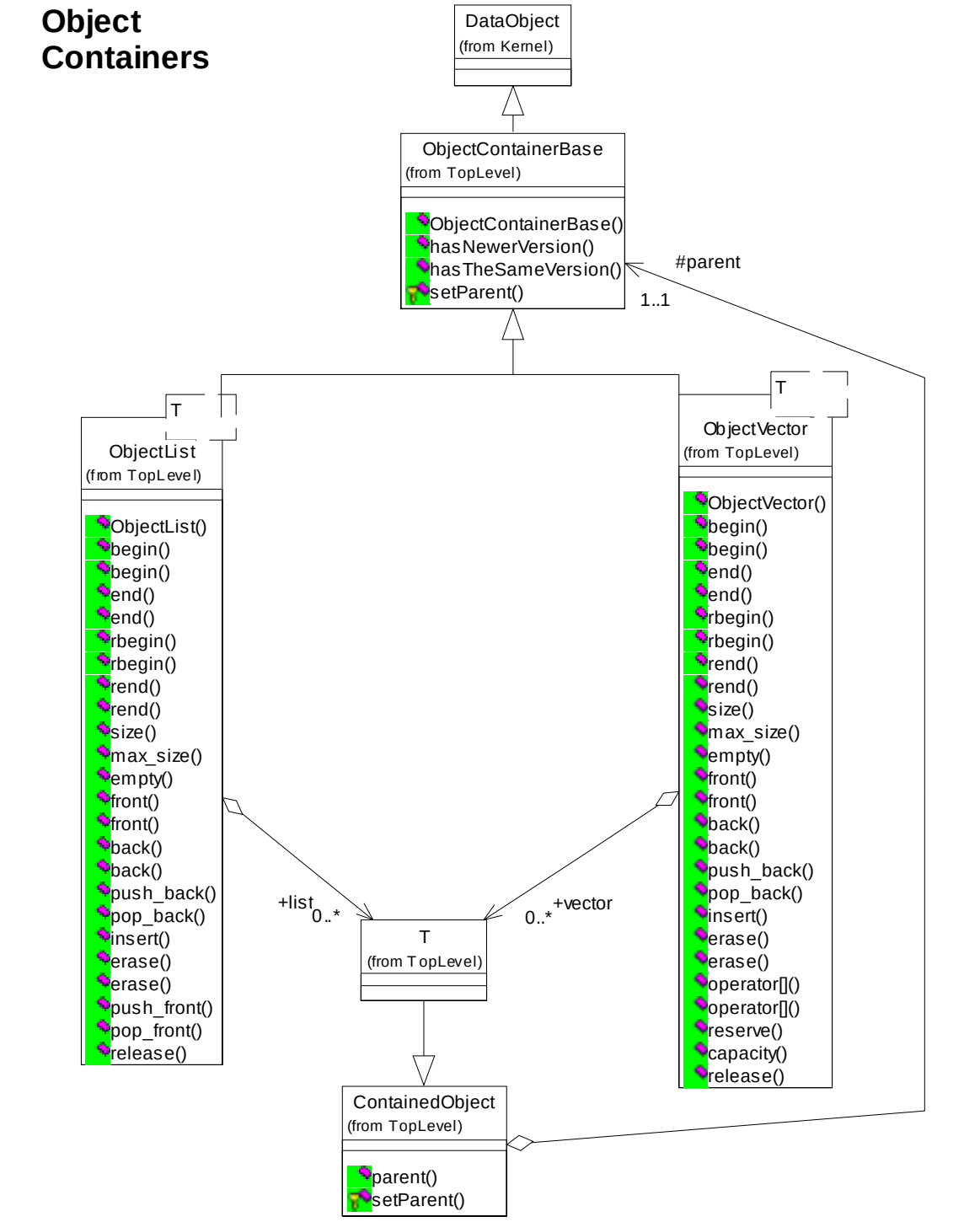


Figure D.3 Object containers for the event model

Index

Symbols

A

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

W